TECHNISCHE UNIVERSITÄT MÜNCHEN

Fakultät für Informatik
Lehrstuhl für Bioinformatik

Incremental Linear Model Trees on Big Data

Andreas Hapfelmeier

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. M. Bichler
Prüfer der Dissertation:	
	1. Univ.-Prof. Dr. B. Rost
	2. Univ.-Prof. Dr. St. Kramer Johannes Gutenberg Universität Mainz

Die Dissertation wurde am 27.07.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 17.05.2016 angenommen.

Ich versichere, dass ich diese Dissertation selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 10.07.2015

Abstract

The data revolution has a similar deep impact to the society as the industrial and the digital revolution. Due to the digital revolution computer were more and more integrated into daily life. Consequently, data volume was steadily increasing. It is assumed that the increase of data volume will even be, for the time being, exponential. This is due to new concepts as, e.g., social networks or the Internet of Things and new data formats to be stored (e.g. video). Big Data is the most well known term describing this phenomenon. Besides challenges in transferring (I/O bottlenecks) and storing this massive amount of data volume, an important task lies in the interpretation and analysis. For the task of knowledge extraction and exploitation, machine learning approaches are well accepted and used. Nevertheless, normal (batch) algorithms are not capable to handle Big Data and are especially not designed to be applied to data streams. Consequently, Big Data systems and algorithms have emerged for this task. The usual Big Data approach is to use increased storage and processing power. This is costly and under specific situations not feasible (e.g. algorithms running directly on sensors or small machines). For this purpose, efficient machine learning algorithms are needed which can process Big Data even in a constrained environment (by time, main memory and storage space). One class of these efficient algorithms are incremental linear model trees (ILMTs), which are still a young field of active research. Their behavior is neither well studied on massive datasets nor on high-speed data streams. Therefore, this work, for the first time, systematically compares the performance of ILMTs on massive stationary datasets under equal conditions. The usefulness of different parameter settings are tested for each ILMT on several data sources. Results give useful insights on the choice of the appropriate ILMT algorithm and its parameter setting. Furthermore, this thesis introduces the incremental pruning approach GuIP as an extension of ILMTs. Although pruning is standard in batch learning, in the incremental online setting, model trees are often only adjusted or pruned if concept drift occurs. For stationary datasets, where no concept drift occurs, no pruning would appear. This results in overly large trees where wrong split decisions are not corrected. Overly large trees are time consuming and hinder the application of ILMTs on high-speed data sources. Evaluation results show that GuIP significantly reduces tree size and runtime, while the prediction error spans from a reduction to a marginal increase with increasing dataset complexity. The third problem tackled in this thesis is that examples from data streams may arrive at a higher speed than the learning algorithms can process. As a consequence, examples have to be dropped and can either not be used for further model improvement or, worse, predictions cannot be made on these examples. To avoid this the data stream processing system PAFAS is introduced. It guarantees a prediction for all unlabeled examples promptly after arrival time, while the model is still constantly improved. Compared with two other processing systems, algorithms in the PAFAS framework showed better prediction accuracy in the majority of cases. In summary, this thesis provides new insights to ILMTs on stationary Big Data. It introduces for the first time a systematic ILMT comparison under equal conditions and a new pruning approach, further improving the applicability of ILMTs. Furthermore, a new data stream processing framework is introduced, improving the usage of incremental learning algorithms on high-speed data streams.

Zusammenfassung

Die Datenrevolution hat eine ähnlich starke Auswirkung auf die Gesellschaft wie die industrielle oder die digitale Revolution. Aufgrund der digitalen Revolution wurden Computer immer mehr in das tägliche Leben integriert. Folglich hat sich auch das Datenvolumen stetig erhöht. Zurzeit wird angenommen, dass der Anstieg des Datenvolumens sogar exponentiell verlaufen wird. Dieser wird durch neue Konzepte wie zum Beispiel soziale Netzwerke oder das Internet der Dinge und aber auch durch neue zu speichernde Formate (z.B. Videodateien) verursacht. Big Data ist der wohl bekannteste Begriff der dieses Phänomen beschreibt. Neben Herausforderungen im Transfer (I/O Engpässe) und der Speicherung dieser massiven Datenvolumen liegt eine wichtige Aufgabe in deren Interpretation und Analyse. Zur Wissensextraktion und -verwertung werden Algorithmen des maschinellen Lernens seit langem angewendet. Normale (batch) Algorithmen sind jedoch nicht in der Lage, Big Data zu verarbeiten und außerdem nicht dafür konzipiert, auf Datenströmen angewendet zu werden. Deshalb sind neue Big Data Systeme und Algorithmen entstanden. Der übliche Big Data Ansatz ist, den Speicher und die Prozessierungskraft zu erhöhen. Das ist kostenintensiv und unter bestimmten Umständen nicht umsetzbar, zum Beispiel, wenn Algorithmen direkt auf Sensoren oder kleinen Maschinen laufen sollen. In diesem Fall sind effiziente maschinelle Lernverfahren notwendig, welche Big Data auch unter beschränkten Ressourcen (Zeit, Hauptspeicher und Speichergröße) verarbeiten können. Eine Klasse solcher effizienter Algorithmen sind inkrementelle lineare Modellbäume (ILMTs), welche erst seit kurzem aktiv erforscht werden. Ihr Verhalten ist weder auf massiven Datensätzen noch auf hoch frequenten Datenströmen genügend untersucht worden. Deshalb vergleicht diese Arbeit zum ersten Mal ILMTs systematisch auf massiven Datenmengen unter gleichen Bedingungen. Der Nutzen verschiedener Parametereinstellungen wird für jeden ILMT auf mehreren Datenquellen getestet. Die Ergebnisse geben nützliche Einsichten in sowohl die Wahl des passenden ILMTs, als auch dessen Parametereinstellung. Des Weiteren stellt diese Arbeit den inkrementellen Pruning-Ansatz GuIP als eine Erweiterung für ILMTs vor. Obwohl Pruning Standard im Batch-Lernen ist, werden Modellbäume im inkrementellen online Ansatz meist nur angepasst oder beschnitten, wenn Konzeptdrift auftritt. Für stationäre Datensätze, welche keinen Konzeptdrift aufweisen, würde kein Pruning angewendet werden. Die dadurch entstehenden Bäume sind übermäßig groß und falsche Splitentscheidungen werden nicht korrigiert. Folglich wird mehr Rechenzeit benötigt und somit die Anwendung der ILMTs auf Hochgeschwindigkeitsdatenquellen verhindert. Ergebnisse der Evaluierung zeigen, dass GuIP die Baumgröße und die Laufzeit signifikant reduziert, während der Vorhersagefehler mit steigender Datensatzkomplexität von einer Reduktion bis zu einem marginalen Anstieg reicht. Das dritte in dieser Arbeit aufgegriffene Problem ist, dass

Beispiele aus Datenströmen in einer höheren Geschwindigkeit eintreffen als sie die Lernalgorithmen verarbeiten können. Als Konsequenz werden Beispiele verworfen und können entweder nicht verwendet werden, um das Modell weiter zu verbessern oder, im schlimmeren Fall, stehen sie nicht für Vorhersagen zur Verfügung. Um das zu umgehen, wird das datenstromverarbeitende System PAFAS vorgestellt. Es kann für alle ungelabelten Beispiele eine Vorhersage zeitnah nach deren Eintreffen garantieren, während das Modell weiterhin regelmäßig verbessert wird. Vergleiche mit zwei weiteren datenstromverarbeitenden Systemen zeigen, dass Algorithmen im PAFAS System in der Mehrheit der Fälle bessere Vorhersagegenauigkeiten aufweisen. Zusammenfassend gibt diese Arbeit neue Einsichten in ILMTs auf stationärem Big Data. Sie stellt zum ersten Mal einen systematischen ILMT-Vergleich unter gleichen Bedingungen und den neuen Pruning-Ansatz GuIP, welcher die Anwendbarkeit von ILMTs erhöht, vor. Des Weiteren wird das neue datenstromverarbeitende System PAFAS eingeführt, welches die Nutzung von inkrementellen Algorithmen auf hochfrequenten Datenströmen verbessert.

Acknowledgements

All over the years, I have met many interesting people, I could share my ideas with and who, on the other hand, inspired me. I want to thank all of them. To be more specific, I want to thank all of my bachelor and master students for their work and their trust (Guokun Zhang, Yassine Azyrit, Christina Mertes and Noelia Ruiz), and all my colleagues at the lab. It was a pleasure to work with you.
I want to especially thank four persons with the most impact on my work. First of all many thanks to Dr. Jana Schmidt. For all the discussions, support and encouragements. You always managed to show me the light... I also want to thank Prof. Dr. Bernhard Pfahringer for his great support during his visit in Munich. All the discussions helped me a lot to make big steps forward. Special thanks to my supervisor Prof. Dr. Stefan Kramer who motivated and inspired me to further go into the area of Machine Learning and Data Mining and who made my PhD position possible. Last but not least, I want to thank Prof Dr. Burkhard Rost for all his support in the final stages.

Contents

Chapter 1

Introduction

1.1 Motivation

> We're drowning in information and starving for knowledge.
> - Rutherford D. Rogers, University Librarian at Yale University, 1985

We are living in a century of knowledge. Information is collected at a huge amount and frequency, with the great aspiration to extract new and, hopefully, world changing knowledge from it. The information collecting process was first started by the libraries: They collected and stored books, periodicals and documents for the public. Already in 1985, Mr. Rogers, one of the world‘s most respected librarians, was worried that more information is produced than anybody could ever read or adopt. Since then, the amount of information growth has increased drastically due to advanced digital techniques and the establishment of the world wide web. Nowadays, library information is further expanded by audio and video material, and physical items, e.g. books, periodicals and documents, are gradually digitized. Taking the US Library of Congress as an example, the approximate amount of available material via the internet is about 74 terabytes for about 15.3 million digital items (of a physical collection of about 142 million items) in 2009 [113] and 235 terabytes in April 2011 [97]. But information and knowledge is not only collected in and distributed through libraries. Since the development and propagation of the world wide web and the technological achievements, information is available for everyone at any place of the world at any time. Humans, as well as computers, are connected worldwide and information is collected, distributed and stored at a high scale. Networked sensors are, apparently hidden, integrated in everyday life. The human environment (e.g. houses, clothes, mobile phones, vehicles) are more and more equipped with sensors and intelligence to create and communicate information (Internet of Things [8] / cyber-physical systems [22]). For example, an airline jet collects 10 terabytes of sensor data for every 30 minutes flight and gas turbines of type 8000H generate from 1500 up to 5000 signals per turbine per second. Humans easily interact and share information from everywhere at any time via computers or devices. In 2010, e.g., 5 billion mobile phones were in use world wide. Smartphones, the main information providers, covered 12 percent of all mobile phones with a growing penetration at more than 20 percent a year [97]. Communication via social networks (Facebook[1], Twitter[2], etc.) became very popular, producing constantly huge amounts of data. For example, 30 billion pieces of content were shared on Facebook every month [97]. File sharing and storing offerings in the cloud (e.g. Dropbox[3], Box[4], Google Drive[5] and SkyDrive[6]) are very popular and frequently used. Megaupload, for example, stored 28 petabytes of user data in 2012 [82]. Companies are collecting trillions of bytes of information about their operations, suppliers and customers. The McKinsey Global Institute (MGI) estimates that enterprises globally stored more than 7 exabytes of *new* data on disk drives solely in 2010 [97].

Over the years, several studies were initiated to get a feeling for the total amount of data generated, stored, and consumed in the world [95, 56, 51, 52, 53, 54, 55, 17, 124, 65, 97]. Although the methodologies and definitions were different and therefore their results vary, all agree on a fundamental point: The data volume in the world is expanding rapidly with an exponential grow within the next future. Figure 1.1 exemplary shows the size of the 'digital universe'(a measure of all the digital data created, replicated, and consumed in a *single* year) from 2010 with a forecast to 2020. New challenges arising with the increasing amount of data are concentrated under the buzz word 'Big Data'. The definition of Big Data slightly differs.
McKinsey defines Big Data as the reference

> '[...] to datasets whose size is beyond the ability of typical database software tools to capture, store, manage, and analyze.'[97]

Gartner recently updated its definition to the following:

> 'Big data are high volume, high velocity, and/or high variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization.'
> - Douglas, Laney. Gartner. June 2012. [37]

[1]http://www.facebook.com; Dec. 2013
[2]http://www.twitter.com; Dec. 2013
[3]http://www.dropbox.com; Dec. 2013
[4]https://www.box.com/; Dec. 2013
[5]https://drive.google.com/; Dec. 2013
[6]https://skydrive.live.com; Dec. 2013

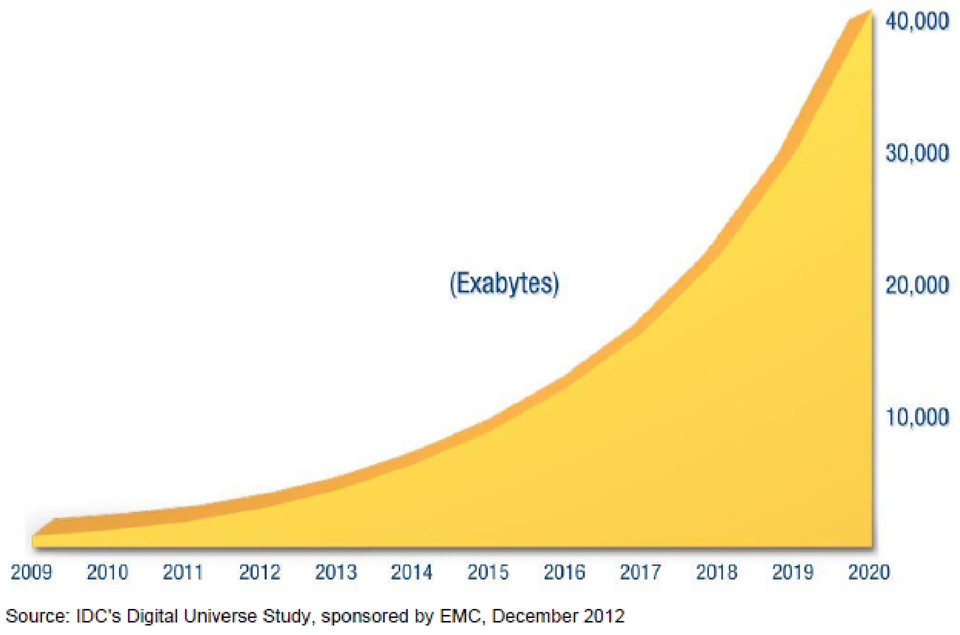


Figure 1.1: The Digital Universe: 50-fold growth from the beginning of 2010 to the end of 2020 (Source: [55])

This definition shortly reviews the main challenges arising with Big Data that are also known as the 4 V‘s:

- **Variety**: Data and information can be available in a structured, semi-structured or unstructured form and has direct impact on the complexity of storage and knowledge extraction.
- **Velocity**: Data retrieval, data processing and knowledge extraction can underlie different speed requirements – batch, near time, real time, streams.
- **Volume**: The pure amount of data itself needs new approaches for storage and processing.
- **Veracity**: Quality and provenance of received data can be diverse (good, bad, undefined, inconsistent, incomplete, ambiguous)

Therefore, a fundamental question is, how to make use of Big Data. In the following two different approach domains for Big Data are presented. First of all, widely used, classic approaches are shown which are handling Big Data under potentially high resources. Next, approaches are introduced for handling Big Data under limited resources.

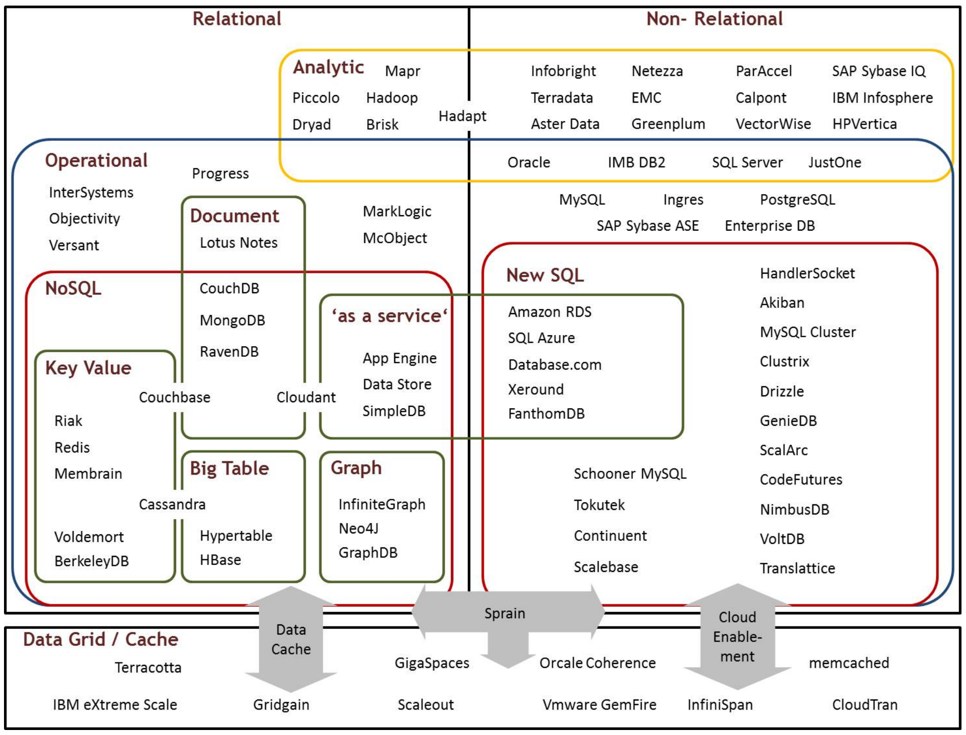


Figure 1.2: The Big Data ecosystem of different systems (Source: [9])

1.1.1 Processing Big Data - Under Plenty Resources

To store, process and visualize Big Data, different approaches and systems have been developed. A surely not complete overview of different systems / products in the Big Data ecosystem can be found in Figure 1.2. There is no best system for all use cases. Choosing the adequate system depends on the circumstances and the goals that should be reached. For example, what kind of data should be stored and processed (structured, semi- or unstructured), at which speed, etc. (see 4 V's).

Relational databases improved their performance on structured data mostly by parallel processing. A famous relational database management system for Big Data was developed by TERADATA[7], but there are also solutions from Microsoft, Oracle or IBM.

Another way to improve the calculation speed on Big Data is to use the main memory more broadly. SAP HANA[8], for example, is the most prominent in-memory database and platform. It offers a performance gain through combined software and hardware techniques. On the software side,

[7]http://www.teradata.com; Dec. 2013
[8]http://www.saphana.com; Dec. 2013

a hybrid (column and row oriented) in-memory database approach is chosen with additional algorithms optimized for in-memory processing. This is additionally supported by replacing the main memory by the CPU-Cache and the data storage on the hard drive by a main memory storage. This reduces significantly the access time.

As conventional relational databases are not flexible and fast enough to store a huge amount of incoming data, while still enabling reads, new systems have been developed. These systems are NoSQL databases (e.g. Cassandra[9], CouchDB[10], MongoDB[11], Redis[12], Riak[13], Neo4J[14], and FlockDB[15]), which are high performance non-relational databases. Their performance is achieved by trading away the power of a relational database: the ACID rules are mostly not guaranteed, resulting in processing advantages. On the downside, packages can be lost and NoSQL databases only guarantee an eventual consistency. Other performance boosts are that no table schema is needed and the system has a horizontal scaling. That means that parallelization is possible by adding new nodes to gain higher performance. Furthermore, column or row oriented database versions are available to achieve the additional performance tip.

A famous system to store and handle unstructured data is the Hadoop system. It is an open source software project that enables distributed processing of large data sets across clusters of commodity servers. It enjoys great popularity through the following user-friendly facts:

- Automatic parallelization, coordination and distribution of calculations (jobs)
- Error tolerance in the face of hardware and software failures
- Automatic load spreading
- Optimization of network and data transfer
- Status and monitoring messages to overview the system
- Cheap and easy expandability

By horizontal scaling, additional processing power as well as storage space can be added to the system. This is done by integrating new nodes (commodity servers). Consequently, processing power and storage space can be cheaply extended without the need for changes to the existing system. The

[9]http://cassandra.apache.org/; Dec. 2013
[10]http://couchdb.apache.org/; Dec. 2013
[11]http://www.mongodb.org/; Dec. 2013
[12]http://www.redis.io/; Dec. 2013
[13]http://docs.basho.com/riak/latest/; Dec. 2013
[14]http://www.neo4j.org/; Dec. 2013
[15]https://github.com/twitter/flockdb; Dec. 2013

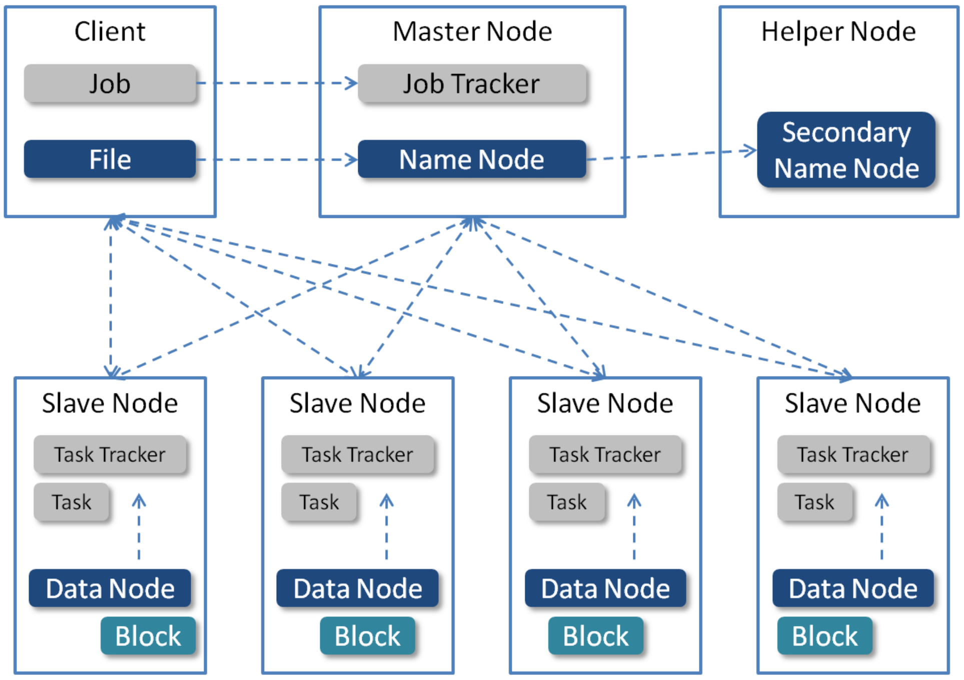


Figure 1.3: Exemplary Hadoop architecture. Blue blocks are used for data storage and gray blocks for job processing.

file system of Hadoop is based on the Google File System (GFS [59]) developed in 2003 and is called HDFS (Hadoop Distributed File System). HDFS is a highly available file system, optimized for very large data amounts. Usually, few but large data files are stored in data blocks with a typical size of 64 MByte. These data blocks are stored redundantly (at least three times) in the file system and are usually only written once and often read.
The architecture of a Hadoop system can be seen in Figure 1.3. The data blocks are redundantly distributed over the so called Slave Nodes (or Data Nodes in the perspective of data storage) and, by that, an error tolerance can be guaranteed. To coordinate the redundancy and to store the location information of the data blocks, a Master Node (Name Node) is needed. For backup reasons, a Helper Node (an additional Secondary Name Node) can be used in the system. Each node represents a server or virtualization. If the system is now queried by a job, transmitted by the user / client, the code is transferred to the Master Node, where the Job Tracker is activated to supervise the job. As the Master Node knows where the necessary data is located, the code is further transferred to the Slave Nodes, the calculations start on each data block in parallel and the results are finally combined. Task Trackers in the Slave Nodes overview the tasks and return status messages to the Job Tracker. The Job Tracker itself overviews the process of the

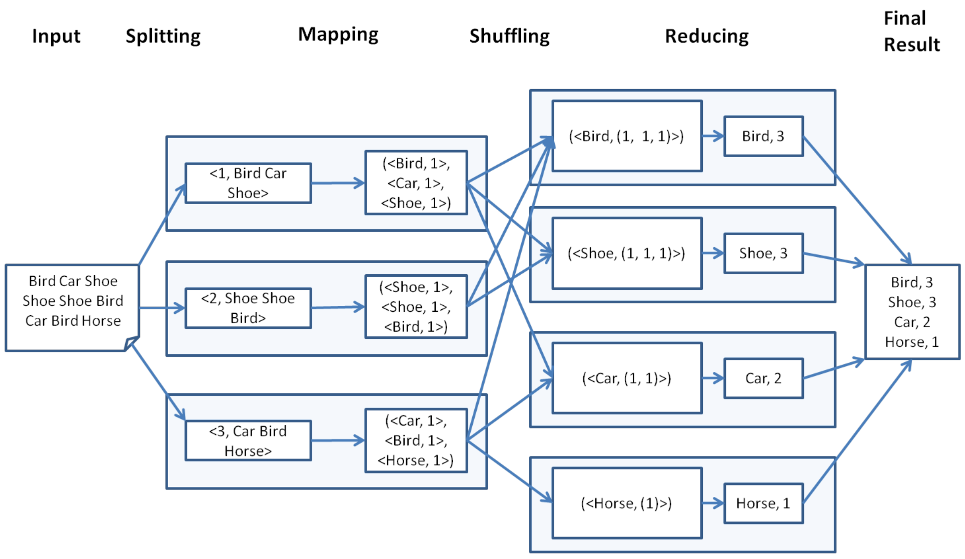


Figure 1.4: Simple MapReduce example: Word count

whole job. It knows the status of each slave node: which results returned manifold and which are still missing. Missing results due to errors can then be recalculated on other slave nodes. Transferring the job (code) to the data, instead of the, usually used, other way round, is a key feature of the Hadoop system. Consequently, the resources used for data transfer can be reduced, which are high in the case of Big Data.
Dividing the data to several slave nodes, enabling parallel processing and merging the information to a final result is known as the MapReduce approach, which was originally developed by Google in 2004 [32]. In the Hadoop framework, the complexity of that system is hidden to the user as he only needs to define a Map and a Reduce step to start the calculation. In the following, the steps of the MapReduce approach are explained by the example shown in Figure 1.4. The task in this example is to count the words of a given file. To do so, the file is split and redundantly stored in the HDFS. In this example, the file is split and stored line by line. The next step is the Map step. This step is the first step, that has to be defined by the user. As input, the step takes a list of key-value pairs ($List(< Key_1, Value_1 >)$). In this example, the line number is the key and the line text is the value. Then, within this step, the user defines some projections, filtering and transformation and returns a new list of key-value pairs ($List(< Key_2, Value_2 >)$). Here, each key is a word in the line and its paired value is the number of occurrences. Now follows the shuffling step, which is provided by Hadoop itself. Identical keys over all Slave Nodes are combined and handed over to the Reduce step. Depending on the number of available Slave Nodes for

the Reduce step calculation, more than one key has to be transferred to one reduce step. In detail, each Slave Node with a Reduce job receives a list of key-value pairs. This time, the value itself is the value list of all keys of this type ($List(< Key_2, List(Value_2) >)$). In our example, each key is passed to one Reduce job with a list of the number of occurrences of the word in each line. The next step, the Reduce step, has to be defined by the user. Depending one the user code, the information is now reduced / aggregated to a new list of key-values pairs, where the values are again single entries ($Lists(< Key_3, Value_3 >)$). In our example, the number of occurrences of each key / word is simply summed up and returned. The calculations from each Reduce job are now collected, aggregated and written back to the HDFS.
To sum up, it can be said that Hadoop scales out rather than scales up and that data throughput is by far more important than access time. Hadoop is not designed for real-time or low latency queries. It is designed to process indefinitely huge amount of unstructured data. To make Hadoop more user friendly and to open new application areas, the Hadoop ecosystem now consists of a huge amount of additional programs. As the amount is steadily increasing, the following list can be only an excerpt:

- **HBase**[16]: Hadoop column database; supports batch and random reads and limited queries
- **Zookeeper**[17]: Highly-Available Coordination Service
- **Oozie**[18]: Hadoop workflow scheduler and manager
- **Pig**[19]: Data processing language and execution environment
- **Hive**[20]: Data warehouse with SQL interface
- **Mahout**[21]: Machine learning and data mining algorithms

Many companies as e.g. IBM, Cloudera or Hortonworks adjusted Hadoop and additional programs to commercial products with further support.

Big Data is often associated with gaining new insights and information which would not be available with less data. This is known as generating Smart Data. Analyzing Big Data leads to Smart Data, which is done more and more with machine learning and data mining algorithms. They are implemented in additional packages and adopted to the specific system techniques. For example, for TERADATA, TERADATA Aster is available and

[16]http://hbase.apache.org/; Dec. 2013
[17]http://zookeeper.apache.org/; Dec. 2013
[18]http://oozie.apache.org/; Dec. 2013
[19]http://pig.apache.org/; Dec. 2013
[20]http://hive.apache.org/; Dec. 2013
[21]http://mahout.apache.org/; Dec. 2013

for the Hadoop system, Mahout is available as an add-on, implementing algorithms for recommendation mining, clustering, classification and frequent itemset mining.

Overall, it can be said that the answer to Big Data seems to be twofold: using more memory and / or multiprocessing. This approach is feasible, as the needed hardware is steadily becoming cheaper. Data storage, for example, is cheap. All worlds music data in 2011 can be stored on a disk drive bought for \$ 600 [97]. Nevertheless, buying and maintaining the hardware is still expensive in the dimensions of Big Data. The additional specialized software is another big burden. For all those who cannot afford the step to buy several servers, to build a Hadoop cluster, or to use servers with huge amounts of RAM, cloud services are cost efficient alternatives. For example, AWS (Amazon Web Services)[22] or Google Cloud Platform[23] offer a variety of possibilities to process Big Data and to extract information. The service is paid for the time used and for the data volume stored and transferred. For short term projects, where data is only collected and analyzed for a short period, using such a service could be profitable. Unfortunately, when a data management and analysis project is planned over a longer period, an own system still seems to be mandatory.

1.1.2 Processing Big Data - Under Limited Resources

Processing and handling Big Data with the afore mentioned methods is not always possible due to the needed resources. Either because the needed resources are too expensive or not available in the area of application. In general, two application areas are possible to handle Big Data (see Figure 1.5): processing the data directly at the origin of creation (e.g. sensors) or on a server structure where the data is sent to and combined (area of classical methods).
For the first application area, consider the Internet of Things. Especially there, more and more sensors are applied to daily products for environment measurements and controls. Companies invest in sensor applications for their products or in companies doing so. The most recent example is the buyout of Nest Labs Inc. by Google [74]. Nest develops intelligent devices for home control, as, e.g., learning thermostats or protection systems with smoke and CO alarms. Fortunately, this trend is not only reserved to big companies as Google. Several systems as the RasperryPi[24] or Arduino[25], to mention just a few, make it possible for every hobbyist to build cost-efficient sensor systems to produce data. This new generation of devices is not only able to send the data to servers, which are collecting and processing the

[22]http://aws.amazon.com; Dec. 2013
[23]https://cloud.google.com/; Dec. 2013
[24]http://www.raspberrypi.org/; Dec. 2013
[25]http://www.arduino.cc/; Feb. 2014

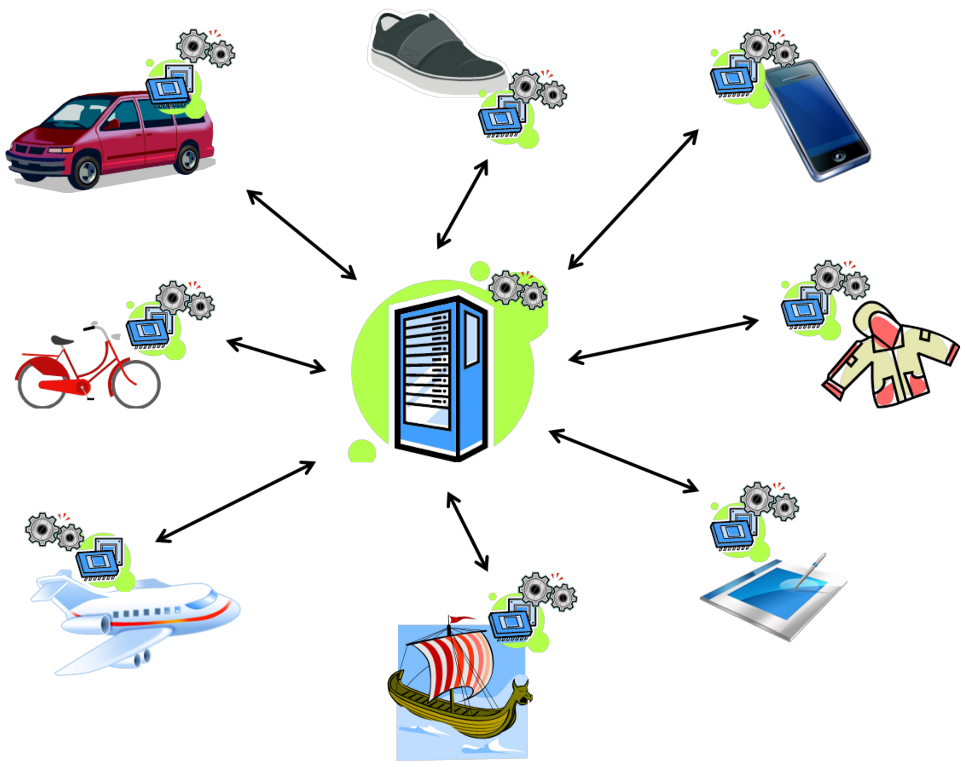


Figure 1.5: Internet of Things – Daily devices are equipped with sensors and processors. Data can be processed directly on the devices or / and sent to a global server.

data from several devices. They also have integrated processing and storage units, making them to small computers that are able to process their own sensor measurements and to gain knowledge from that. By intelligent data analysis through machine learning and data mining tasks, information and actions can be directly derived and alarms or counter actions can be initiated. As the systems only have up to 1Gb of RAM and one or two processors of low speed, the arriving data is already by definition [97] Big Data for such systems. Consequently, very resource efficient algorithms are needed to process the data. The classical Big Data methods are not applicable under these constraints.
Even on the server side, the second application area, where all data from different sources is collected, combined and where knowledge is extracted, efficient knowledge extraction methods are needed. The available resources should be used efficiently or even be reduced. Classical Big Data processing approaches need a huge amount of resources, resulting in high maintenance and energy costs. To reduce these kinds of costs, the new challenge is to

handle Big Data in a responsible way. This trend is known by *green computing.* Hardware costs can almost be neglected in comparison to the energy costs accumulated over time. Constantly huge amount of energy is needed for running the needed environment. This is further increased with processing / computing requirements. Consequently, algorithmic efficiency plays a main role in green computing.

A group of algorithms working very well under limited resources are incremental learning algorithms. They are fast and memory efficient learning algorithms designed for online learning tasks on data streams. Due to their ability to learn fast under a given amount of memory, they can be ideally used for machine learning or data mining tasks on sensor devices itself. Additionally, due to their efficiency, they can be considered as algorithms for green computing. While developed for stream processing, they can be deployed on massive datasets as well, showing all the benefits for green computing. Although the domain of incremental learning algorithms is yet a relatively young research area, it shows a high research activity. Unfortunately, even if there is great research, only small effort is put into creating reusable algorithm code. Fortunately, there are two open source projects collecting incremental learning algorithms in a toolkit. The first one is the VFML (Very Fast Machine Learning) framework[26], initiated by Pedro Domingos and Geoff Hulten and written in C and Python. It has a collection of important learning algorithms for mining high-speed data streams and very large datasets. By using its API, new algorithms can be easily developed. The second toolkit is the MOA framework[27], written in Java and developed at the University of Waikato. It is under active development and has a collection of learning algorithms supporting stream classification, stream clustering, outlier detection and recommender systems. New algorithms can be easily added to the existing system.

1.2 Thesis Outline

This thesis concentrates on the second domain of Big Data processing approaches: algorithms working under limited resources. A class of efficient algorithms which could be applied in this domain are incremental linear model tree algorithms. The focus lies on the application of incremental linear model tree algorithms on massive data sources: massive datasets as well as data streams with massive examples.
This thesis is organized as follows:
Chapter 2 gives an introduction to online and especially incremental learning. The incremental learning process is explained and differences to the batch learning approach is shown. Next, model evaluation approaches are

[26]http://www.cs.washington.edu/dm/vfml/; Dec. 2013
[27]http://moa.cms.waikato.ac.nz, Dec. 2013

introduced for the batch as well as for the online setting. At last, an exemplary incremental algorithm WINNOW is explained in detail.
Chapter 3 explains decision trees in detail. From tree fundamentals to the explanation of the most important algorithms, everything is explained for an understanding of regression and classification trees in the batch and the incremental setting.
Chapter 4 gives an overview of the data sources used throughout this thesis.
Chapter 5 shows, for the first time, a systematic performance evaluation of incremental linear model trees which are not well studied on massive stationary datasets. The evaluation is performed under equal conditions in three different dimensions: prediction error, running time, and memory consumption. The performance evaluation tests the algorithms within the same framework on large-scale artificial and real-world datasets, under various parameter settings.
Chapter 6 introduces GuIP, a new pruning approach for ILMTs with approximate lookahead based on the actual prediction error of the models. It is an extension of incremental linear model trees with approximate lookahead in general and is exemplary integrated into the FIMT algorithm. GuIP can be used to produce smaller trees, which increases their processing speed and consequently, favors their application on high-speed data sources.
Chapter 7 will focus on the problem of high-speed data streams in combination with processing-speed limited learning algorithms. High-speed data streams can overwhelm the learning algorithms by delivering examples faster than the algorithms can process them. As a consequence, examples have to be dropped and can either not be used for further model improvement or, worse, predictions cannot be made on these examples. Missing predictions are, e.g., a problem in emergency systems where prompt predictions are needed on each example. Otherwise, severe consequences could arise. Consequently, frameworks are needed to make the best use of the examples arriving. For this purpose, we introduce a new framework called PAFAS which guarantees a prediction for all unlabeled examples promptly after arrival time, while the model is still constantly improved.
The thesis closes with Chapter 8, a summary of this thesis and an outlook on future work.

Chapter 2

Online and Incremental Learning

Machine learning and data mining algorithms are used to automatically extract knowledge from data. Most of the algorithms were once developed on only few examples, due to data shortage at the time when the fields were emerging. Consequently, the developed algorithms were designed to store all training examples in main memory for processing (batch approach), which was possible back then. Ever since, the data amount, as well as the data availability has changed. Data is now abundantly available, resulting in huge amounts of data to be processed, or it is constantly growing over time by receiving new examples from data streams [3]. Big Data solutions have been developed to target this challenge (see section 1.1). But the disadvantage of these solutions are the amount of required resources. Solutions are needed to process large amounts of data and data from data streams under limited resources. Online learning algorithms, processing examples on the fly and updating their model incrementally after each example, are one kind of these algorithms.
This chapter gives an overview of the principles of online learning and introduces incremental learning. First, the learning process is shown and differences to the batch approach are highlighted. Second, methods for online model evaluation are shown. Then, concept drift is introduced and finally, an exemplary incremental learning algorithm (Winnow) is explained in more detail.

2.1 Learning Process

The conventional learning approach for batch learners (see Figure 2.1) is based on two assumptions:

- All training data is available at once at training time

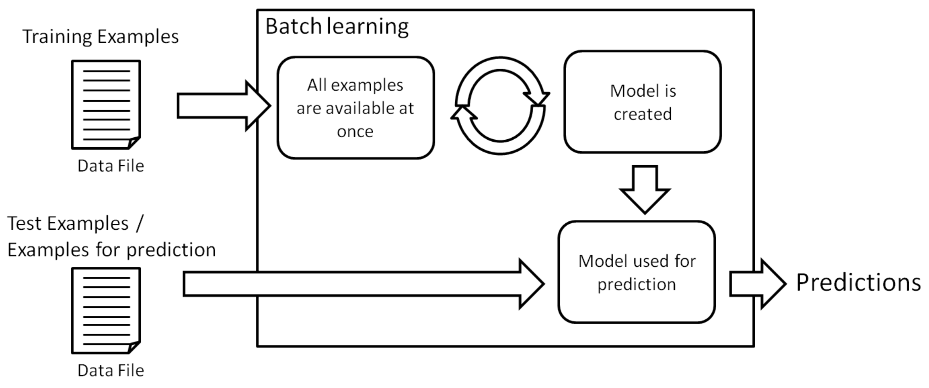

Training Examples
Data File
Batch learning
All examples are available at once
Model is created
Test Examples / Examples for prediction
Data File
Model used for prediction
Predictions

Figure 2.1: Batch learning approach

- The main memory is large enough to store all data during the learning phase

If these assumptions hold, the training examples are loaded in the main memory and are used on the learning algorithm to build the model. Additionally, loops can be created to show the training examples several times to the learning algorithm to constantly improve the model on the examples. Nevertheless, it can be assumed that for Big Data, the amount of available memory is too small for devices with limited resources. Training examples have to be stored and several data processing steps have to be performed in the main memory. To be able to create the desired model with the given restricted resources, techniques are needed to reduce the amount of examples. Two areas of research for this purpose are sampling and load shedding techniques. Sampling is a method to draw examples from a distribution or dataset under specific conditions. Common methods are for example: rejection sampling, importance sampling, Gibbs sampling or slice sampling [15], to mention just a few. Load shedding, on the other hand, reduces the example load by discarding unprocessed examples and is often used in the domain of network analysis.
All these approaches have one thing in common: examples are ignored which carry potentially important information for the algorithm. Even if the best is done to identify the most important examples, memory can still be too small to store all important examples, or not all important examples are found. Consequently, an efficient approach to integrate all examples in the model building process should be preferred.
Another drawback of batch algorithms is their problem to handle data streams. In the online setting, new examples are constantly arriving from data streams. Batch algorithms need their training examples at once to calculate the model. An exemplary workflow of a wrapper approach, using

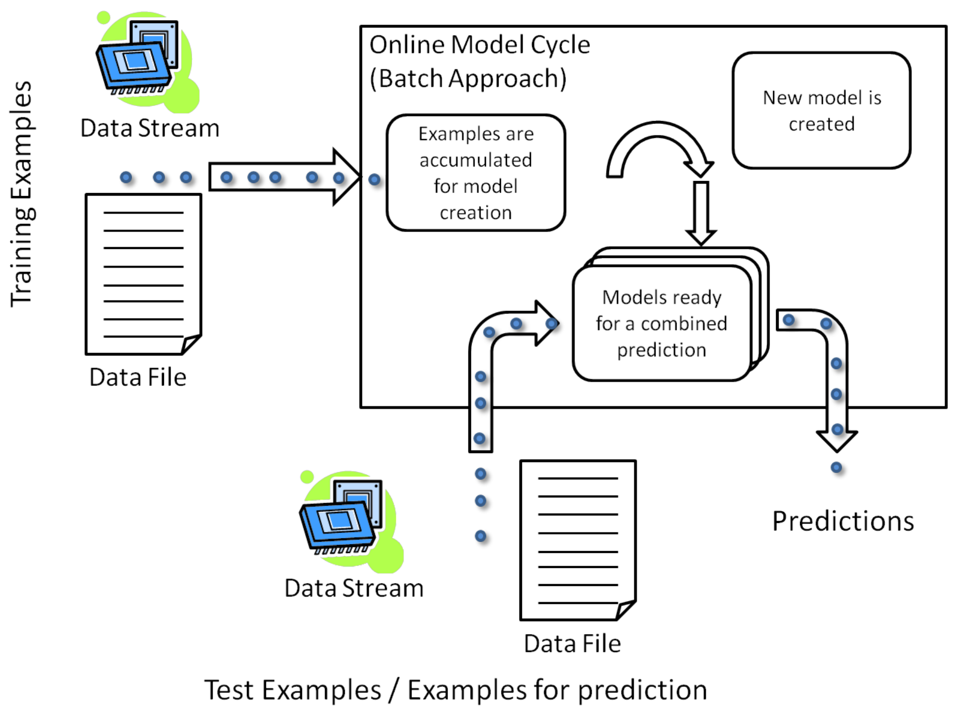


Figure 2.2: Online learning with a batch approach

a classical batch algorithm for online learning, can be found in Figure 2.2. New examples are constantly coming in from the data stream. Then, the model that was created by the batch algorithm on the examples so far, cannot be updated solely by the new examples. But, if all the data from the data stream can be stored, e.g., if the data stream is limited or the data storage is unlimited, a model can be periodically induced from all available data stored so far. The periodicity is strongly dependent on the necessity of recalculation and the induction time of the batch algorithm. In most cases this setting does not hold. Storage space is somehow limited and data streams are unlimited. Thus, there will always be the time point where there is too much data for the storage space. For this setting, a window based approach can be used, collecting a specific amount of new incoming examples. From these examples in the window, a model is induced by applying the batch algorithm. Over time, several models are built, which can be combined for prediction. One possibility for this is to introduce a weight for each model in the prediction, to favor specific models. This could be that models having a higher prediction accuracy could be stronger weighted or models created on more recent examples. An advantage of this window based approach is that only the data for the current learning batch has to be stored. This has an storage advantage in comparison to storing all exam-

Table 2.1: Online algorithm main requirements

Requirement 1	Process one example at a time, and inspect it only once
Requirement 2	Use a limited amount of memory
Requirement 3	Work in a limited amount of time
Requirement 4	Be ready to predict at any time

ples. Nevertheless, it is a challenge to determine the window size with the correct amount of data. By using too big window sizes, the batch learning algorithms could take too much time to induce the model and the model induction on the next, newly arrived data window is delayed. This may produce a chain reaction: Examples which are not yet used for learning, but are now too old for the current data window, have to be dropped, or a much higher storage amount is needed to store all examples not yet processed. Setting the window sizes too small will result in a high number of models, which will increase the storage size, but can be fixed by deleting the oldest models. This may impact the combined prediction power, which could be overall weak due to the reduced amount of training examples during each model induction. With too few examples during the training process, the induced model may be poor at generalizing to new examples, which results in higher prediction errors. An overall disadvantage of a wrapper approach is that the memory management can only be done on a per model basis as classical batch algorithms are used. A more fine granular memory management could be necessary. Exemplary wrapper approaches can be found by Wang et al. [139], Street and Kim [126] and Chu and Zaniolo [28].
A more convenient possibility to learn on data streams is to create new learning algorithms adjusted to the new real-time requirements posed by the data stream setting. There are four main requirements an online algorithm should meet to properly learn on data streams (see Table 2.1). To meet these requirements on data streams, incremental algorithms are developed for efficient learning. A schematic example for the application of incremental algorithms on data streams (as well as on big datasets) is shown in Figure 2.3. As data is coming in, possibly fast, from data streams, the learning algorithms have to adapt the current model to the new information by requiring only a limited amount of time before the next example arrives. Consequently, only one example is processed at a time and is discarded afterwards[1] (Requirement 1). Furthermore, the algorithm has no control

[1]Under normal conditions, the examples cannot be used for several iterations due to strong memory and processing time restrictions and have to be discarded after updating the model once. But, in principle, it would be possible to store some examples in the short term for further refinement, as long as requirement 2 is met. Also redirecting a stream to use it a second time would be possible. However, in practice, these options are normally

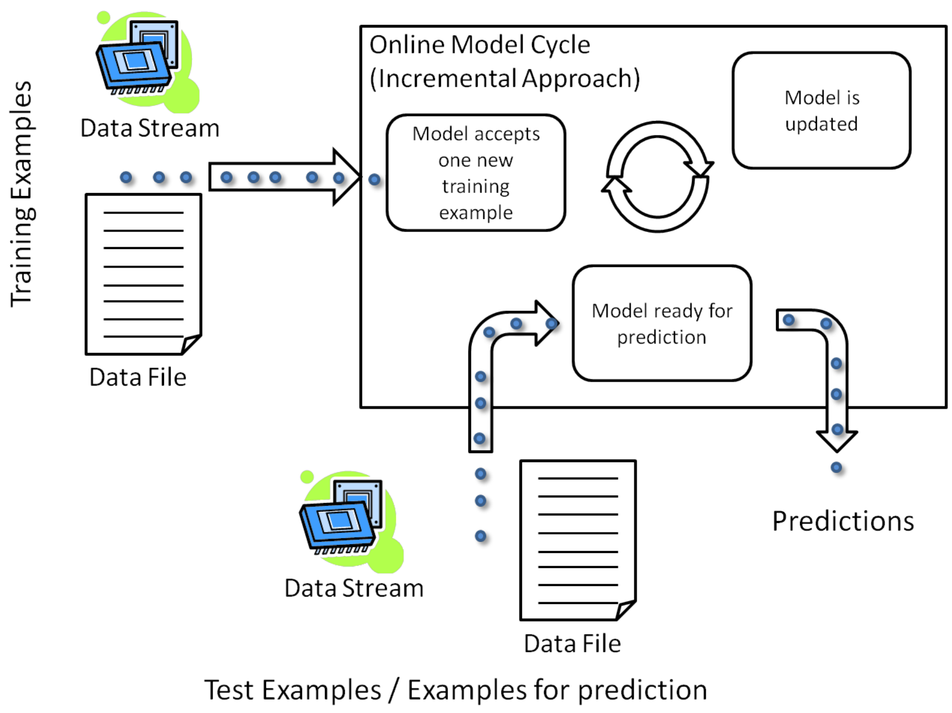


Figure 2.3: Online learning with an incremental approach

over the order of the observed examples and has to process the examples in the order they are coming in. During the whole process, only a limited amount of memory is available (Requirement 2), which has direct impact on the learning process. Due to the huge amount of data which has to be processed by the learning algorithm and the memory constraint, not all examples observed so far can be stored. Only limited and abstract information representing the examples can be stored. This is a hard constraint for the online algorithm update process and one of the main differences to the batch algorithm. This challenge is usually solved by storing some statistics over all observed examples. These statistics consist of aggregated values over all examples which can be incrementally updated and, for that, are memory efficient. The model induction process is based on these statistics and can then also be updated.
The model update process has only a specific maximal time to finish (Requirement 3), as new examples are steadily coming in. The algorithm has to be ready to accept the next example as soon as it comes in. Otherwise the example is lost or has to be temporarily stored on secondary storage. Chapter 7 discusses in more detail the impact of data stream speed on learning

not possible.

algorithms. Even though the time between the incoming examples can vary, most learning algorithms cannot benefit from that fact. It would be beneficial if the learning process could be performed stepwise by updating the model over several stages. The more time the algorithm has before the next example comes in, the more time can be spent for the model update process and the better it becomes. As most learning algorithms are not supporting this concept, the main agreement is that the update process should be as fast as possible. At least as fast as the smallest time difference between the possible shortest example arrivals, so that no example will be lost.
Once a model has been induced from training examples from the stream, it should always be ready to make predictions on new unlabeled examples (Requirement 4). The learning algorithm should have induced the best possible model from the data seen so far and provide it for prediction. As new examples are steadily coming in, the learning algorithm constantly updates the algorithm, which should be no drawback for the prediction. Quite the contrary, after seeing more examples, the algorithm should further improve the model to gain better predictions. Consequently, changing models and the subsequent predictions on different model states poses special new challenges to the model evaluation process, discussed in the next section.

2.2 Model Evaluation

To evaluate the current performance of an existing model, or to compare different algorithm approaches and to find the best model, performance measures of the models can be compared. Therefore, three different performance measures are mainly of interest:

- **Algorithm speed:** Evaluates how fast the algorithm can process examples. The training speed is often of main interest, but the time needed for new predictions is important as well.
- **Memory usage:** Evaluates how much memory is needed by the algorithm during the learning process and how much space is needed by the final model.
- **Prediction error:** Evaluates how well the induced model is able to predict the target variable.

While the error measure seems to be the most important one in the batch setting, all three measures are equally important for the online setting. Even the most correct algorithm is useless if it cannot be applied due to too high memory consumption or because it is too slow. Although this also holds for the batch setting, it is more severe in the online setting with its strong restrictions. While the measurement of memory consumption and processing speed is straightforward, the error of the induced model can be measured in

Table 2.2: Confusion matrix of the classification task

		Predicted Class	
		C_1	C_2
Actual Class	C_1	true positives (TP)	false negatives (FN) - type II error -
	C_2	false positives (FP) - type I error -	true negatives (TN)

Table 2.3: Error measures for the classification task

Sensitivity	$\frac{TP}{P}$
Specificity	$\frac{TN}{N}$
Precision	$\frac{TP}{TP+FP}$
Accuracy	$\frac{TP+TN}{TP+FP+FN+TN}$
F-Measure	$\frac{2TP}{2TP+FP+FN}$

different ways. In the following, different error measures are presented for the classification and the regression task. Afterwards, techniques are shown to correctly estimate the performance measures, by introducing common approaches for the batch setting and adaptations developed for the online setting.

2.2.1 Error Measures

The error measure shows how well a target value y is fitted by the model prediction $\hat{y}$.
For the classification task, the target value is a class value (a value from a possibly infinite set, mostly two values). The classification error is normally based on the confusion matrix (see Table 2.2), and consequently, on the number of true positive (TP), false positive (FP), true negative (TN) and false negative (FN) examples. Different error measures can be calculated based on these numbers, to evaluate the classification model (e.g., sensitivity, specificity, precision, accuracy and F-measure; see Table 2.3). Additionally, the decision can be supported by visualizations. Prominent examples are the Recall-Precision-Chart and the ROC-Curve.
For the regression task, where a numerical value is predicted, it is not sufficient to use error measures judging if the correct target value has been predicted or not. Evaluation measures are needed taking into account the distance of the predicted value $\hat{y}$ from the correct value y. The most com-

Table 2.4: Error measures for the regression task

Absolute error	$y_i - \hat{y}_i$
Squared error	$(y_i - \hat{y}_i)^2$
Mean absolute error	$\frac{\sum_{i=1}^{d} \|y_i - \hat{y}_i\|}{d}$
Mean squared error	$\frac{\sum_{i=1}^{d} (y_i - \hat{y}_i)^2}{d}$
Relative absolute error	$\frac{\sum_{i=1}^{d} \|y_i - \hat{y}_i\|}{\sum_{i=1}^{d} \|y_i - \bar{y}\|}$
Relative squared error	$\frac{\sum_{i=1}^{d} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{d} (y_i - \bar{y})^2}$

monly used error measures for the regression task, are shown in Table 2.4. Detailed information on evaluation measures can be found in several publications and books (e.g. [142, 61]).
The thoughtful reader might have recognized that the above mentioned error measures can represent the correctness (e.g. accuracy) as well as the error (e.g. absolute error) of the predictions. All over this thesis, error measures are seen from the error perspective. Consequently, an increase of an error measure results in an increase of the error (e.g. absolute error) and a decrease of the correctness (e.g. accuracy).

2.2.2 Performance Evaluation Techniques

To find the best fitting model, the performance measures have to be applied in a way, that the results are reliable. Testing the error with the same dataset the model has been learned on (the training set), is not valid. The achieved results would be too optimistic, as they do not represent the error on new, so far unseen, examples. Consequently, the performance of the model has to be tested on unseen examples by the learning algorithm during the model induction. This group of examples is called test set. Another complication in the learning process is overfitting: the learning algorithm tries too hard to explain the training data. By that, the model weakens its generality, the ability to perform good predictions on yet unseen examples, by overadapting to the training data. Consequently, the performance on the test set will be poor. To avoid these and other problems, evaluation techniques have been developed which are introduced in the following. To show the necessity of adapted techniques for the online setting, commonly used techniques for the batch setting are shown for comparison in advance.

Batch Techniques

Techniques for the batch setting were originally designed to optimally use a limited amount of data for the training process. As data was available at once, but very limited in size, techniques were needed to use as much examples for the training process as possible and yet to give valid performance evaluation results. In the following the most common evaluation techniques are shown. Further information can be found in the literature [61, 142, 86].

Holdout and Random Sampling The *holdout* method is the intuitive approach of using one part of the available dataset as training set and the other as the test set (holdout set). To avoid biases in the distribution of the target and feature values in the newly created sets, the examples are randomly selected from the overall set. Typically, two thirds of the overall examples are randomly chosen in the training set and the remaining third is forming the test set. Other combinations are possible as well. The learning algorithm is now executed on the training set to induce the model. The performance of the induced model is then tested by predicting the target of all examples in the test set and calculating error measures as mentioned above. The main criticism of the holdout method in the batch setting is that a large amount of examples are only used for testing instead of improving the model by adding them to the learning set. Consequently, the estimates of the error measures are assumed to be pessimistic.
As the estimations can vary greatly depending on how the test and training set is divided, the *random subsampling* method tries to approximate the actual error measures by repeating the holdout measure k times. Each time the examples in the training and test set change due to the random partitioning. Consequently, for each of the k runs, performance measures are retrieved and a final score can be found by averaging over all runs.

Cross-validation The cross-validation method is the next step to optimize the example use for the training and testing. In *k-fold cross-validation*, the data is randomly partitioned into k mutually exclusive and approximately equally sized subsets (folds) $F_1, F_2, ..., F_k$. Similar to the random subsampling method, training and testing is performed k times. Each time, $k-1$ folds are used for training and the remaining fold is used as the test set. This means that in the first run, F_1 is used as test set and the remaining folds $F_2 - F_k$ are used in combination as the training set. In the second run, F_2 is used as test set and $F_1, F_3 - F_k$ as the combined training set. This is continued over all k runs. Performance measures are obtained by dividing the sum of the performance measures over all runs by the total number of examples. In contrast to the random sampling approach, each example is used the same number of times $(k-1)$ for training and exactly once for testing, reducing the introduced bias. Nevertheless, an imbalanced class

distribution between the folds can still occur by this approach. *Stratified* k-fold cross-validation tries to cope with the class distribution by creating a class distribution in each fold similar or hopefully identical to the one of the entire dataset. On top of that, the cross-validation method can be additionally performed several times, each time with another random partitioning of the folds, enabling the calculation of the variance of the performance measure. Most commonly, ten-fold cross-validation is recommended [86], but the results should still be treated with caution [18].
The *Leave-one-out* method is a special case of the cross-validation approach. The principle is pursued to its extreme: k is set to the amount of examples in the dataset and, consequently, each of the k folds contain only one example. In each run, the maximal amount of training examples is used by still getting a performance evaluation with one example. A valid estimation of the error measure can only be achieved by taking the results over all runs into account. This method has two main benefits: As many examples as possible are used for the learning process, and random effects by fold splitting are avoided. But these benefits also come with some disadvantages: It needs much more time and computing resources than the k-fold cross-validation method, and stratification is not possible. Consequently, it is possible to construct scenarios where the methods fails in measuring the generalization power.

Bootstrap The Bootstrap method [38] samples the training examples from the whole dataset, in contrast to the methods already shown, with replacement. The most commonly used Bootstrap method is called *0.632 Bootstrap* and creates a *bootstrap sample* (training set) of the size of the whole dataset, by sampling with replacement. By that, on average, 63.2% of the original examples will end up, maybe several times, in the training set. Examples not chosen for the training set are used in the test set (approx. 36.8% of the original examples). Performance measures are estimated by combining estimates on the training as well as on the test set (0.632 x $Measure_{Test}$ + 0.362 x $Measure_{Train}$) to compensate the lack of unique training examples. Further reliability of the measures can be gained by repeated random runs. While this method works well on very small datasets, this method also has its critics [86].

Online Techniques

The online setting has other requirements to the performance measure validation process than the batch setting. While for the batch setting the focus is on the optimal data usage due to example shortage, data can be considered as abundant available in the online setting. Especially test sets can be large without having the problem of sacrificing training examples. Consequently, the concerns of repeatedly reusing the examples from the dataset in training and test sets to make the most of the limited available data, is not a concern in the online setting. There, the following new challenges are mainly in focus and have to be solved by the online evaluation techniques:

- **Model development:** Online or incremental models are constantly developed with every training example from the data stream, used to hopefully increase the model quality. Consequently, there is no static final model available which can be evaluated with a test set to achieve the performance measures. Instead, the model has to be evaluated over time to see the development of the performance measures with an increasing number of training examples. From these time courses, extrapolations can be made during the development phase of the algorithm to estimate the performance on a possibly unlimited amount of training data. In the application phase of the algorithm, when the model is used in a live scenario, the performance is constantly available and counter investigations can be initiated if the performance drops.

- **Concept drift:** Online algorithms are normally used on data streams where the examples are arriving over time. Consequently, a time dependency is introduced which can be of importance in the evaluation phase. An increasing error during evaluation must not be because of wrong decisions of the learning algorithm. It can also be a hint for a concept drift in the data (further explained in Section 2.3). If concept drift changes are not handled by the learning algorithm itself, it should be tested or evaluated on the data sources in advance or in parallel. Otherwise, the error increase over time could be for a number of different reasons.

- **Memory limitation:** Running under limited memory resources is an important focus in online learning. Constantly calculating the performance measures, in the development as well as in the application phase, to judge the model performance, needs at least temporal memory. Intermediate and final results have to be stored and, with certain evaluation techniques as the holdout method, test examples have to be stored as well. Consequently, the memory has to be shared between the model learning and the evaluation process. This could raise some challenges under strongly limited resources.

- **Time limitation:** Online learning algorithms should be as fast as possible to be able to process all examples from the stream. Evaluation techniques need additional time, depending on the complexity of the used evaluation method. This time needs to be available to meanwhile evaluate the learning algorithm. In the development phase, interrupting the data stream to evaluate the model is a common procedure. During the application phase, this approach is not possible as the examples are still continuously coming in. Stopping the process for an evaluation task would result in unprocessed examples which are lost forever in the worst case. These examples could, e.g., be outliers which would have been of great interest. Consequently, fast evaluation methods, not affecting the algorithm processing speed too much, are needed in the application phase. In the development phase, more time can be spent for the evaluation under certain circumstances.

- **Example order:** The performance results are strongly dependent on the order of the examples coming in. Applying the algorithm on the same stream at slightly different times will result in, hopefully only marginal, different performance measures. This is due to the fact, that different examples or examples in different orders are available for the learning process. To get an overview of the learning algorithm performance on any example ordering, performance measures should be averaged during the development phase over several runs with different example orderings. By additionally showing the variance over the runs, the influence of the example order on the learning algorithm can be estimated. During the application phase, this fact has no impact on the evaluation process as there is only one example order on which the performance is needed.

For the online setting, mainly two performance evaluation methods exist: the *prequential* and the *holdout* evaluation [14, 13], which are explained in the following in more detail.

Prequential The prequential, or also called Interleaved Test-Then-Train, method [31] constantly evaluates the model. Each example arriving from the data stream is first used as a test example to evaluate the model and then as a training example to further improve it. The prediction error of the current example is added to the sequentially accumulated error measure over all test examples seen so far. This value can now be used in the development phase to compare different runs or algorithm approaches, or in the application phase, to monitor the model performance. The advantage of this method is, that only a small amount of additional memory is needed to store the accumulated performance measures. Additionally, if the testing method is fast enough, this approach is very useful to monitor the model quality in the application phase as the process has not to be stopped for a long testing phase. The downside of this approach is that the learning curve of the prequential error measure is known to be a pessimistic estimator. It suffers from potentially large errors committed during the early phases of training. To reduce or eliminate the influence of such early prediction errors, fading factors [50] or sliding windows can be used on the prequential error measure. These methods are especially useful in the application phase where the current performance is of importance. Another disadvantage of this approach is that it is hard to accurately measure the training and testing times, as they can be hardly separated.

Holdout The holdout evaluation method is similar to the one already discussed for the batch setting. Examples from the data stream are selected in a separate holdout test set. This test set is used to evaluate the model and to retrieve the performance measures. As there is no final static model in the online setting, the model performance has to be repeatedly tested with the holdout test set. By this approach, the development of the performance measures can be observed over time. In contrast to the holdout approach in the batch setting, using examples for the test set is not harmful for the algorithm quality, as examples are abundantly available. The holdout test set is formed by fetching a predefined number of examples from the data stream prior to training. This holdout test set can now be used to repeatedly test the model quality and new examples arriving can be used to further improve the model. This approach has the advantage that varying estimates between different test sets are avoided and it is valid as long as there is no concept drift in the data. In the case of existing concept drift, the model would be updated with the examples of the changing or finally the new concept, while the stored holdout set still consists of examples from the old concept. This would result in a wrong performance evaluation. To cope with concept drift, the holdout test set can be used as a queue prior to the learning algorithm. Equivalent to the FIFO principle, new examples from the data stream are stored in the holdout test set queue and with every example coming in, the example stored for the longest time is released. This example is then transferred to the training algorithm to further improve the model. This approach has the advantage, that the current performance of the model can be tested without the influence of early made mistakes. On the downside, this approach needs additional memory to store the holdout test set. Depending on the size of the test set and the type of information stored, this could be too much for devices with strongly limited resources. Additionally, the repeatedly performed test phases need the time to make predictions for all examples in the holdout test set. During the development phase, the data stream can be repeatedly stopped and holdout evaluation tests can be performed. During the application phase, this approach would result in example loss as the data stream can not be stopped. More advanced approaches as cloning the model for the holdout evaluation in the application phase are imaginable, but under limited resources, these approaches are highly challenging or impossible.

Realization of evaluation methods As online learning itself is still a relatively young field of research, the evaluation techniques in this matter are not yet well studied. Recent work [14] analyzed several publications [35, 49, 48, 75, 104, 126, 43, 28] in the field of online learning to get an impression of the quality of the currently used evaluation techniques. The main findings are described as follows: Memory limits are not in the focus of the publications. Mostly no explicit memory limitations are placed on the algorithms. Consequently, the impact of limited memory on the overall performance is mostly not well studied. Furthermore, most algorithms are trained on datasets with less than one million examples. Only some use up to 10 million examples and tens of millions examples are used only rarely. This is a problematic fact, as online algorithms should be developed to run on an indefinite amount of examples. Extrapolating the performance from a couple of examples to an unlimited amount is problematic. This usually gives only a very limited insight to the useability of the algorithms on unlimited data streams. Performance measures as time, memory and accuracy can change significantly and learning curves can still cross after substantial training has occurred [83]. An exemplary evaluation of this topic can be found in Chapter 5. Most publications use the holdout method as evaluation technique during the development phase. Calculated performance measures are mostly based in a single holdout run, making it impossible to quantify the uncertainty of the results based on, e.g., confidence intervals. Reruns due to the example ordering are often not performed with the note that the method is not highly order sensitive and results would not significantly change. Unfortunately, this is mostly not proved. Exceptions to the above mentioned, are available throughout the considered publications, where one can find more detailed analyses.

This overview shows that, in contrast to the evaluation methods in the batch area, the requirements for a valid evaluation of online algorithms are not yet realized by the scientists. They are still on the way to find a common understanding, or a so called gold standard, of the validation process of online learning algorithms. For Richard Kirkby, to mention an example, a learning algorithm on data streams is only adequately evaluated, if it is tested on large streams of hundreds of millions of examples and with explicit memory restrictions. In his opinion, any less than this would not test the algorithms in a realistic setting [83].

2.3 Concept Drift

Massive amounts of data can be collected in two ways: Either by fetching the information at once at one specific time point or by collecting and accumulating it over time (e.g. datasets achieved from data streams). Each dataset has its own underlying information dependencies and logic. With introducing a time dependency in the dataset, *evolution of data* can appear. The logic can evolve over time illustrating different challenges from logic change recognition to logic change handling. If it can be assumed that there is no logic change over time within the dataset, as e.g. when the dataset is collected at once, the dataset is called *static* or *stationary*, otherwise *evolving*. As stated by Gao *et al.* [57], there are three different possibilities that the data distribution $P(x, y) = P(y|x) * P(x)$ can evolve over time. First of all, changes in $P(x)$ can occur, also known as *virtual concept drift*, *sampling shift* or *distribution / sampling change*. Additionally, changes in the conditional probability $P(y|x)$ or in both, $P(x)$ and $P(y|x)$ are possible. Changes in the conditional probability are also known as *concept drift*. Depending on the speed of change, the concept drift can be further divided into *gradual concept drift* and *abrupt concept drift*. Sometimes, the abrupt concept shift is also called *concept shift* [14]. Furthermore, concept drift is either *local* or *global*. A local concept drift happens only over a limited set of ranges for a sub-sample of the measured attributes, while a global concept drift targets all possible values of all attributes, including the target variable. Evolution of data is interesting from the following two perspectives:

- Finding and visualizing concept changes in the dataset can give valuable insights. Understanding the chances in the data can have financial as well as knowledge impact. Further analysis or direct counter measures can be initiated. Several methods can be used to detect and handle concept change, as e.g., CUSUM (cumulative sum) test [105, 121], Page-Hinkley test [101], GMA (geometric moving average) test [114], statistical tests [12, 10], DDM (drift detection method) [48], EWMA (exponential weighted moving average) [14], velocity density estimation method [2] and KL-distance [30], to mention just a few besides many others (e.g. [60, 80]). Detecting concept change is useful in a variety of domains as, e.g., network traffic monitoring [120], monitoring of gas turbines [11] or cement rotary kilns [89].

- Data evolution has direct impact on learning algorithms. The concept in the data changes over time and differs from the one the model was learned on. Although the model generality should slightly compensate the effect, this harms the model quality. The prediction error will increase and the model should be relearned to adapt to the new concept. One possible approach is to frequently scan the data stream for a concept change. If detected, a new model can be learned on the new concept and replace the old one. Two different methods are commonly used to detect change: Evaluating the performance with a defined set of performance indicators (e.g. accuracy, recall and precision over time [85]) or comparing the data distribution over two different time-windows (e.g. [80]). Instead of relearning the complete model under concept change detection, more sophisticated, model specific methods can be used. The detection method can be directly integrated into the learning algorithm, enabling recalculation of specific parts of the

Algorithm 1 Winnow(Example $e < x_1, ..., x_n, y >$)

```
1: if $\sum_{i=1}^{n} w_i x_i > \psi$ then
2:     $\hat{y} := 1$
3: else
4:     $\hat{y} := 0$
5: end if
6: if $\hat{y} == y$ then
7:     do nothing
8: else if $\hat{y} == 1$ AND $y == 0$ then
9:     $Demotion(w_i)$, if $x_i == 1$
10: else if $\hat{y} == 0$ AND $y == 1$ then
11:     $Promotion(w_i)$, if $x_i == 1$
12: end if
```

model. This saves time and unnecessary calculations. This method should be especially preferred under massive data, when fast processing is needed. Exemplary learning algorithms are CVFDT [67] and FIMT-DD [70].

As this work mainly focuses on stationary data (as data files or data streams), this is only a coarse overview of concept change. Therefore, not all details of the above mentioned literature are explained. More detailed information can be found under the afore mentioned citations.

2.4 WINNOW – An Exemplary Incremental Algorithm

The WINNOW algorithm [90, 91] is a simple and fast incremental learning algorithm developed by Nick Littlestone, similar to the perceptron algorithm [115, 116]. In contrast to the perceptron algorithm, a multiplicative scheme is used with the advantage of performing better with many irrelevant dimensions. It is a simple algorithm scaling well to high dimensional data and consequently, it is an ideal example to illustrate the online learning setting with an existing incremental learning algorithm.
A pseudocode of the learning algorithm [90] is given in Algorithm 1. Each example shown to the WINNOW learning algorithm consists of n Boolean-valued attributes as well as one Boolean-valued target variable $X = \{0, 1\}^{n+1}$. For each attribute x_i, where $i \in \{1, ..., n\}$, the algorithm stores a weight w_i initialized with 1. First of all, the model is tested with the new example e (lines 1-5). A prediction is made by summing up the multiplications of each attribute value with the specific stored weight ($\sum_{i=1}^{n} w_i x_i$). If this value is greater than a given threshold ψ (line 1), the prediction $\hat{y}$ for example e is 1 (line 2), otherwise 0 (line 4). A common value for the threshold is $n/2$. This

prediction can now be compared with the target value y of the example and the error can be added to the error measure to evaluate the algorithm (not shown here, but explained in the sections above). Then the algorithm tries to improve its prediction with the example e by starting the learning task (line 6-12). Depending on the error of this prediction, the algorithm adjusts its weights to improve further predictions. If the prediction was correct (line 6), the weights are well adjusted and no changes are needed. When a type I error occurs (line 8), a demotion step is needed, as the weights have been too high. Consequently, all weights contributed to the result (weights on attributes of the example with value 1) are set to 0 in the first version of Winnow (line 9). On the other hand, too low weights result in type II errors and need a promotion step (line 10-11). All weights contributing to the wrong prediction are now multiplied by α. A typical value for α is 2. After adjusting the weights, the next example is processed by the algorithm. An improvement of the algorithm [91] changed the demotion step to decrease the weights by dividing them by α instead of setting them to 0. While this approach is known as the *positive winnow*, many modifications are available as, e.g., the *balanced winnow* or *snow winnow*.
This short introduction of a simple incremental algorithm should recall the following important facts of an incremental online learning algorithm:

- Each example is observed one by one.
- Examples are first used to test and then to train the algorithm. Changing this order would result in wrong evaluation measures.
- Each incremental algorithm needs to store some statistics over the past examples as examples cannot be stored. In the winnow algorithm, these statistics are the weights w_i. This is a lightweight statistic. More complex algorithms have to store more complex statistics.
- A memory limit is needed and available as the maximal memory consumption is the current example, the stored weights and the intermediate calculations.
- Time is also a strong limitation and the time consumption, for this learning algorithm, is the same for each example. Seeing more examples does not result in longer training times. This behavior is benchmark and often not accomplished by the majority of the online algorithms.

Chapter 3

Decision Trees

Algorithms for the induction of decision trees are a famous and well developed family of learning algorithms. One of the main advantages is the expressiveness of the resulting models. In comparison to other learning algorithms, the output of the model gives the user valid information about the learned concept. Consequently, the user may gain further insights into her or his data. In this chapter, the learning task is explained first. Then, the fundamentals of decision trees are explained. How they are trained, further improved and adapted to the regression and classification task. Knowing the basics, an overview of existing regression and classification algorithms are given for both, the batch and the incremental setting.

3.1 Learning Task

Let a dataset $D_{Set} = \{e_1, \ldots, e_n\}$ or a data stream $D_{Stream} = \{e_1, \ldots, e_\infty\}$ be given. Each example e_i is represented as a vector $< x_{i1}, \ldots, x_{im}, y_i >$. The data contains a variable of interest Y and several descriptive variables X_j. The overall goal in the prediction task is to find the underlying concept to explain the characteristics of the variable of interest Y by the other variables X_j. Error measures extracted from the learning task can be used to show how well the underlying concept could be explained through the learned model. For a regression task, the target variable (variable of interest) Y is a numeric variable and, for that, contains continuous and possibly unlimited values. A classification task is defined by a categorical target variable Y. An example showing two underlying concepts from simple datasets, containing only one descriptive and one target variable, can be found in Figure 3.1 for a regression and a classification task. The examples show the dependency between the number of people on a party and its attractiveness. Usually, every party gets more attractive with an increasing number of participants. When the party gets too crowded, its attractiveness is decreasing again. This fact can be observed for a regression and a classification task

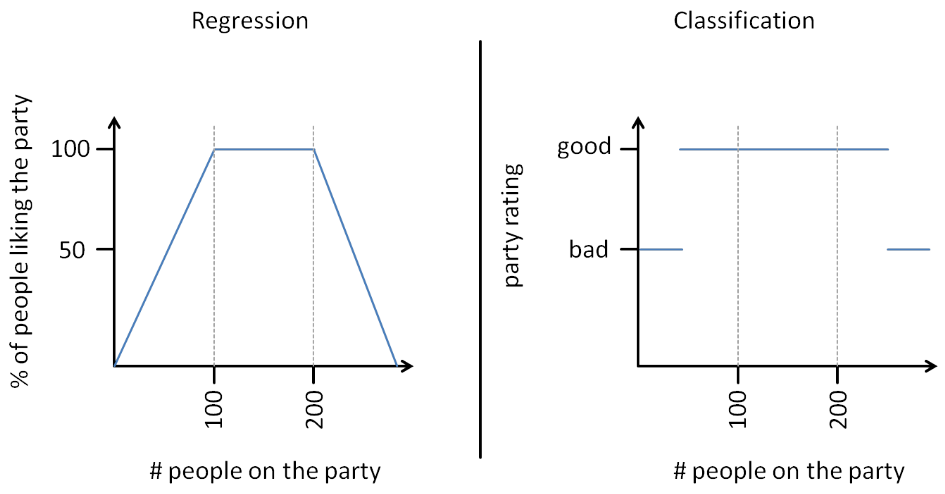


Figure 3.1: The dependency between the number of people on a party and its attractiveness is displayed for a regression (left side) and a classification task (right side). A party is more attractive with more people participating. Too many people on the other hand can turn the atmosphere.

in the example. In both tasks, we want to predict the attractiveness of the party from the number of people participating as the input variable. For the regression task, we want to predict the percentage of people liking the party (continuous target variable) and for the classification task if the party is good or bad (categorical target variable). For both tasks, the specific model should detect the underlying concept:

- **Regression Task:** With increasing number of participants, the party is more liked. The acceptance is linear increasing and the maximal attractiveness is found with 100 participants. This maximal attractiveness is maintained until 200 participants. Including more people results in a linear decrease in the attractiveness.
- **Classification Task:** With less than 50 participants, a party is considered as bad. From 50 to 250 participants, it is good, and with more people it is too crowded and bad again.

3.2 Decision Tree Fundamentals

In the following, the fundamentals of decision trees are shown. First, decision trees are defined. Then, an overview of the induction process is given for batch and incremental learners, followed by an explanation of the prediction process. Then, different splitting criteria are shown in detail for the

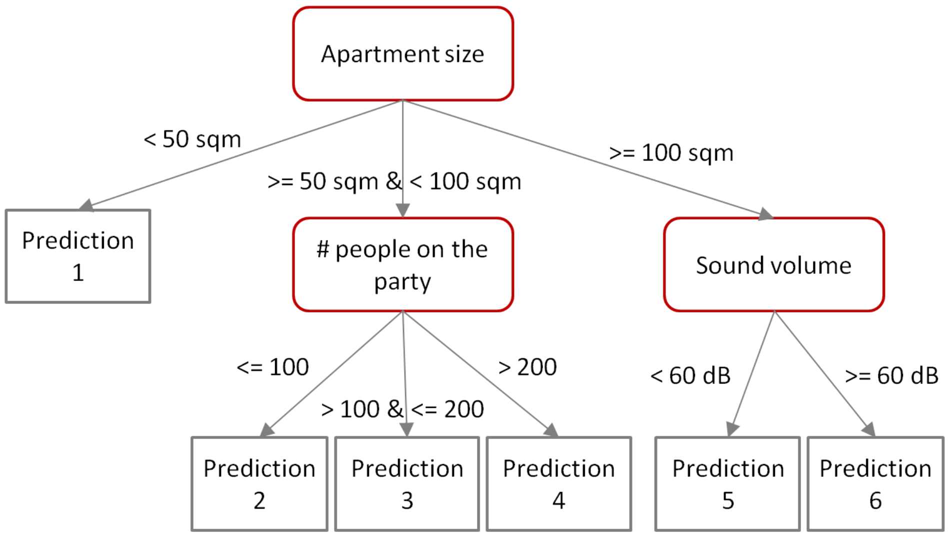


Figure 3.2: An exemplary decision tree based on the party example. Edges, containing the decision thresholds, are leaving the internal nodes (round rectangles) which are displaying the splitting attributes. At the end of each path, the leaf nodes (angled rectangles) can be found, holding some kind of prediction instruction.

regression and the classification task. Finally, the principles of pruning and lookahead are explained.

3.2.1 Definition of Decision Trees

A tree is a directed acyclic graph T that consists of nodes N and edges $E = (n_i, n_j)$, were $n_i, n_j \in N$. A tree is spanned over several levels, beginning with one node, the so called *root node*. Nodes are further divided into *inner / internal nodes* IN and *leaf nodes* LN. Including the root node, each internal node has a splitting decision sd and two or more child nodes, connected by an edge. The splitting decision is based on an attribute X_j and one or several thresholds th of the attribute value on which the input space is partitioned. The specification of the splitting decision can be different, depending on the type of the chosen attribute. Splitting decisions on categorical attributes have thresholds consisting of subsets of the categorical values. For numerical attributes, the thresholds are either specific numerical values or ranges. The splitting decision can produce trees with internal nodes having more than two edges. In the most common case, a binary tree with two edges per internal node is constructed. The last category of leaves, the leaf nodes, have no child nodes and are the last nodes on the path from the root towards the last level. They are used for prediction and, consequently, contain a model m or some other prediction instructions. Depending on the algorithm, internal

nodes can be deployed with a model or prediction instructions as well.
If the decision tree should be used in the online setting, additional properties are needed. In this setting, the decision tree is improved and extended over time with every example arriving. The training data is never available at once and examples should not be stored or accumulated due to memory limitations. However, to base the tree improvement decisions, as e.g. the node splitting decision, on valid information, statistics over the so far seen examples are stored. These statistics are called *split statistics spst* in this work and are stored with each leaf node in the online setting.
An exemplary decision tree is shown in Figure 3.2 and its induction process is explained in the next section.

3.2.2 Induction of Decision Trees

The main idea behind the decision tree induction process is to divide the input space into subregions with stable and good predictions. This is performed by a divide-and-conquer approach. Subspaces are chosen and are further divided to finally get the best division in the perspective of the error measure of the prediction task. To illustrate this, consider Figure 3.1. For the regression task, it would be beneficial to split the input space in the dimension of the number of guests on the party. Dividing this dimension at 100 and 200 would result in three subspaces. For each subspace a linear model should result in low error measures as the ideal splits of the input space have been chosen. A less good choice would be a division in two subspaces by dividing at, e.g., 150 guests. Linear models in each subspace would result in higher error measures. Finding the best splitting decision is part of the splitting and look-ahead strategy and is explained in the next sections. Depending on the availability of the data and if a batch or an online approach is needed, the induction process slightly differs.

In the **batch setting**, all data is available at once and the tree can be learned on the complete dataset. That means that the given dataset is divided into subsets of examples for which the lowest error measures are achieved with adequate models. This is done in an iterative process as shown in Algorithm 2. This algorithm illustrates a general induction method for the batch setting. At each iteration, a node n is created and a splitting criterion is used to find the best splitting decision sd on the available subset of examples. This is done by taking into account already made splitting decisions stored in the splitDecisionList, to avoid resplits (line 2). If there is no beneficial split or all splitting possibilities are already used, splitting is not possible. Consequently, the splitting decision will be empty (line 3), the iterative process is stopped, and the current subspace is not further divided. In that case, node n is considered as a leaf node ln. A model or some kind of prediction instruction is learned on the dataset D and attached to ln, which is then returned (lines 4-5). Otherwise, if there is still an

Algorithm 2 BatchInduction(Dataset D, List splitDecisionList, splittingCriterion)

```
1: createNode $n$
2: splittingDecision $sd$ := splittingCriterion($D$, splitDecisionList)
3: if $sd == NULL$ then
4:     learn model $m$ or prediction instructions on $D$
5:     return N as a leaf node $ln$ including model $ln.m$
6: end if
7: addSplitDecisionToNode($n$, $sd$)
8: splitDecisionList := append(splitDecisionList,$sd$)
9: for each threshold $th$ of $sd$ do
10:     $D_j$ := getSubset($D$, $sd$)
11:     if $D_j == \emptyset$ then
12:         learn model $m$ or prediction instructions on $D$
13:         createChildWithPrediction($n$,newNode, $ln.m$)
14:     else
15:         attachChild($n$, BatchInduction($D_j$, splitDecisionList, splittingCriterion))
16:     end if
17: end for
18: return $n$
```

improvement possible, the found splitting decision sd is attached to the node n and added to the splitDecisionList (lines 7-8). The next step is to further divide the current subset and apply the induction process on the subsets. Consequently, the dataset D is split into subsets D_j by applying sd on D (line 10). The number of created subsets depends on the amount of thresholds th in sd. Please remember, a threshold can be any subset of values for categorical attributes and any value or value range for a numerical value. Each split decision can have several thresholds, depending on the algorithm. Under very special conditions, the application of sd on D can result in an empty subset D_j. In that case, a leaf node is created and added to n with prediction instructions formed on the subset D. This can be considered as the best instructions possible for the subspace (line 12-13). Under normal conditions, the resulting subset D_j is not empty and can be used to further extend the decision tree. A child node is added to the node n by recursively applying the method on the subset D_j.

In the **online setting** the data is not available at once and consequently, the decision tree has to be extended and improved over time. In comparison to the batch approach, there will probably never be a final model to use. With each example arriving in the online setting, the model will change. An exemplary induction pseudocode for a decision tree in the online setting can be found in Algorithm 3. Just as the batch algorithm, a root node is

Algorithm 3 OnlineInduction(Node n, Example $e_i < x_{ij}, ..., x_{in}, y_i >$, List splitDecisionList)

```
1: if $n$ type of internal node then
2:     $in := n$
3:     Node $n_{child}$ = getLeafNode($in$, $in.sd$, $e_i$)
4:     OnlineInduction($n_{child}$, $e_i$)
5: else
6:     // $n$ is a leaf node $ln$
7:     $ln := n$
8:     update model $ln.m$ with $e_i$
9:     update $ln.spst$ with $e_i$
10:    // test for split
11:    if split $ln$ preferable using a $sd \notin$ splitDecisionList then
12:        add $sd$ to splitDecisionList
13:        $in := ln$
14:        attach $sd$ to $in$
15:        createChildNodes($in$, $in.sd$)
16:    end if
17: end if
```

first created and the algorithm is then called for every example e_i arriving (OnlineInduction(root node, e_i)). Beginning with the root node, each node is tested if it is an internal or a leaf node. If it is still an internal node, the example is tested on the splitting decision *sd* located at the internal node, to find the next child node on its path to the proper leaf node (line 3-4). This is iteratively done until a leaf node is reached. Having found the end of the tree path, the example is first used to update the model m or the prediction instructions in the leaf node (line 8). Additionally, the stored split statistics *spst* in this leaf node are enriched with e_i (line 9). *spst* is then used to test if a tree extension is preferable and which *sd*, not yet used on the path, should be chosen (line 11). This *sd* is then stored in splitDecisionList as a decision already used on the path (line 12) and *ln* is split using *sd* (line 13-15). This means that *ln* is transformed to an internal node *in* having *sd* as its splitting decision. Additionally, new child nodes are added to the node. Depending on the chosen splitting decision the number of children could vary.

Additional steps could be possible for the batch, as well as for the online induction algorithm. The above shown algorithms are only general descriptions of the process. Depending on the algorithm, extensions as, e.g., models in the internal nodes or additional statistics could be possible. Furthermore, pruning approaches could be included to keep the tree small and to increase its expressiveness. Pruning approaches are explained in detail in Section 3.2.5. To evaluate the performance of the model, error measures can be calculated on the tree. Different calculation approaches are possible and are

Algorithm 4 Prediction(Node n, Example $e_i < x_{ij}, ..., x_{in} >$)

```
1: if $n$ type of internal node then
2:     $in := n$
3:     Node $n_{child}$ := getLeafNode($in$, $in.sd$, $e_i$)
4:     Prediction($n_{child}$, $e_i$)
5: else
6:     // $n$ is a leaf node $ln$
7:     $ln := n$
8:     Prediction $\hat{y}$ := prediction($ln.m$, $e_i$)
9:     return $\hat{y}$
10: end if
```

extensions to the general induction process. These approaches have been explained in Section 2.2.
Specific decision tree induction algorithms mostly differ in the following aspects [88]:

1. Which type the target variable has (numeric or categorical)
2. Which splitting criterion they use (cf. Section 3.2.4)
3. How the final tree is pruned (cf. Section 3.2.5)
4. Whether they use lookahead strategies to avoid locally optimal trees (cf. Section 3.2.6)

On top of these aspects, online learning algorithms also have to store splitting statistics to be able to perform valid split decisions. These splitting statistics are also algorithm specific and will be discussed with the afore mentioned aspects in the following sections.

3.2.3 Prediction with Decision Trees

In the batch setting, the final learned model can be used to perform predictions on examples where the target variable y is missing (unlabeled examples). In the online setting, there is no final model. The model is constantly improved with every example arriving. However, it should be possible to make predictions on arriving examples at any time. Nevertheless, the process of the prediction task is the same for a model built in the batch or the online setting. A general pseudocode describing the process can be found in Algorithm 4. The example e_i, for which a prediction is needed, traverses the tree from the root node to the leaf (Prediction(root node, e_i), lines 1-4). This is again done by constantly applying the splitting decision of the internal node on the example to choose the appropriate child node. Finally arriving at the leaf node, the prediction instructions are used on e_i to get a prediction $\hat{y}$, which is returned (lines 7-9).

An exemplary decision tree based on the party example can be found in Figure 3.2. Besides the information of how many people are at the party, the information of the apartment size and the sound volume is considered as useful to predict the attractiveness of the party. A possible party example is classified by this tree by starting with the root node, which checks the apartment size where the party is located at. The root node separates the input space of all possible parties into parties in apartments less than 50 sqm, between 50 and 100 sqm, and greater than 100 sqm. If the apartment size of the current party example is less than 50 sqm, the decision tree model would give a prediction based on the prediction instructions "Prediction 1". If it is between 50 and 100 sqm, the example would follow the second edge to the next internal node. This one would now test how many people are on the party. Depending on the amount of people at the current party example, one of the three leaf nodes (Prediction 2, 3, or 4) would be used to predict the attractiveness. For the last path (larger than 100 sqm), the sound volume has been considered as useful to make predictions for the examples. It further separates the input space into two groups, with specific prediction instructions.

3.2.4 Splitting Criteria

In general, a splitting criterion is a heuristic that chooses a splitting decision for an inner node of the tree. This splitting decision is chosen in a way that the prediction tasks in the created subspaces have a lower error. Each possible split has a score representing its error gain. By ranking them, the one with the lowest score can be chosen as the final splitting decision. This section presents some possibilities of splitting criteria. Note that these criteria do not provide a complete survey of split statistics that are used in decision tree induction but only give some basic ideas and commonly used statistics. In the following, splitting criteria for the classification, as well as for the regression task are shown.

Classification Splitting Criteria

In the following, a couple of splitting criteria for the classification task are shown. The majority of the approaches try to increase the purity of the class distribution in the subspaces. An increased class distribution purity naturally increases the prediction task performance, as the distribution is biased towards only a few or one class.

Information gain The information gain [107] is based on an idea of Shannon who proposed a measure for the information content of messages [122]. Basically, the idea behind the splitting criterion is that an attribute which leads to the purest separation or classification of the examples is chosen as splitting attribute. The purest separation is measured with the information gain. The information gain of an attribute X in a dataset D is given by Equation 3.1.

$$IG(X) = I(D) - I_X(D) \tag{3.1}$$

It reflects the difference of the information that is needed to classify an instance before and after a split with attribute X. I(D) (refer Equation 3.2) reflects the expected information that is needed to classify an example in D and is also known as the entropy of D. This is estimated by the class distribution in D: $p_i = \frac{|C_{i,D}|}{|D|}$, where p_i is the probability of class C_i (out of m classes) in dataset D.

$$I(D) = -\sum_{i=1}^{m} p_i log_2(p_i) \tag{3.2}$$

$I_X(D)$ (refer Equation 3.3) represents the amount of information that is still needed for the classification after the split was made with attribute X. X contains v values and thus divides the dataset D into v subsets D_j for which $X = x_j$.

$$I_X(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times I(D_j) \tag{3.3}$$

For each subset D_j, the entropy is calculated and the weighted sum represents the entropy of the split. Over all possible splitting attributes X_j in D, the one with the maximal entropy reduction is finally chosen as the splitting attribute.

While explained for categorical attributes, this measure can also be adapted to continuous valued attributes. A drawback of the information gain is that it prefers attributes with many values. The more individual values an attribute can take, the higher the information gain can be. As an example consider a dataset with a unique identifier as class per example. If this attribute is taken for a split, a perfect classification is achieved and the information gain is maximized. However, such a model is unsuited for the application on unknown instances because it is essentially rote learning and no generalization takes place.

Gain ratio To compensate this shortcoming of the information gain, the gain ratio was introduced. The basic idea is to normalize the information gain to get rid of the bias towards multi-valued attributes [110]. The normalization is defined in Equation 3.4

$$SI_X(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times log_2 \frac{|D_j|}{|D|} \tag{3.4}$$

Again, attribute X has v values and D_j is the set of examples where $X = x_j$. $SI_X(D)$ takes into account the subset sizes compared to the whole data set. Therefore, the goal is not to get the purest separation of the training data but a good proportion of classification error and data set size. The gain ratio is then defined as shown in Equation 3.5.

$$GR_X(D) = \frac{IG(X)}{SI_X(D)} \tag{3.5}$$

One shortcoming of the gain ratio is that it prefers attributes that lead to unbalanced splits, i.e. one resulting partition is much larger than the other one.

Gini gain The Gini index [21] measures the impurity of a dataset D as shown in Equation 3.6, where m gives the number of classes in the dataset D and $p_i = \frac{|C_{i,D}|}{|D|}$ estimates the probability that a tuple belongs to class i.

$$Gini(D) = 1 - \sum_{i=1}^{m} p_i^2 \tag{3.6}$$

The Gini gain measures the impurity reduction of a split on attribute X subdividing dataset D in n datasets D_i. As shown in Equation 3.7, the weighted sum of the impurity of each resulting partition D_i is calculated and subtracted from the impurity of the whole dataset to calculate the Gini gain.

$$GiniGain_X(D) = Gini(D) - \sum_{i=1}^{n} \frac{|D_i|}{|D|} Gini(D_i) \tag{3.7}$$

The splitting decision with the highest Gini gain (the highest impurity reduction) is finally chosen. The Gini gain can also be applied to continuous attributes by considering every possible split point that leads to two subsets. Again, the split point with the best Gini gain is chosen. However, also the Gini gain has its drawbacks. It prefers multivalued attributes and also attributes that result in equally sized partitions.

C-SEP Fayyad *et al.* [42] argued that a measure that is based on the impurity may not be suitable for classification learning where more than two classes are present. Thus they proposed a new attribute selection measure. The basic idea is that small trees should be inferred which cannot be guaranteed by impurity measures. This should be achieved by separating classes as early as possible to keep the final tree small. Therefore, they derive a family of measures C-SEP (class separating) that should be used instead. The group of selection measures that are proposed should fulfill the following constraints concerning the partitions (D_1, D_2) of a dataset.

1. When the classes in D_1 are disjoint with the classes in D_2, the measure is maximal, minimal if the class partition is identical.
2. Instances of the same classes should reside in the same partition.
3. If the class distribution is permuted this results in another measure value.
4. The measure is differentiable, non negative and symmetric with respect to the classes.

This should result in a measure that best separates the classes by each split and moreover, keeps the instances of the same classes together.

An exemplary measure is based on the class vectors v_1, v_2 of the resulting partitions (induced by a variable test r) and their orthogonality and is therefore called ORT measure

$$ORT(r, S) = 1 - cos\theta(v_1, v_2) \tag{3.8}$$

where $cos\theta(v_1, v_2)$ is based on the usual definition of angles between two vectors (their dot product).

MDL principle This measure computes the coding size (number of bits) for the decision tree encryption and the error description (errors that were induced by the tree). In the attribute selection process, the attribute resulting in the smallest coding size is chosen as the splitting attribute. A split is omitted if it would increase the coding size and the induction process is stopped in this leaf node. It is stated that the attribute selection using the MDL principle has the least bias towards multi-valued attributes [87].

Multivariate splits Another method to select a good splitting decision is to consider several attributes for a split [102]. That means that not only one attribute is used in a splitting decision. Several attributes decide together about the branching of the tree [134]. One possibility for multivariate splits is to use linear discriminants as a partition function in each internal node. Therefore, soft entropy can be used to calculate the best parameters for internal coefficients. DT-SEPIR uses this kind of splitting criterion [76]. Multivariate splits may prevent decision trees from becoming too large and thus, not understandable. Decision trees are for example hard to read if continuous attributes are repeatedly tested along one branch of the tree or if subtrees are copied within the tree. Moreover, multivariate splits can also be considered as another form of feature construction. For example, the CART system can use linear combinations of features for splitting. Although the resulting trees are smaller, they are harder to interpret due to the feature combination and also take longer to be learned. Thus the user always has to balance the drawbacks and merits of multivariate splits.

Regression Splitting Criteria

The main difference in tree learning for the regression task is that the predicted variable is numeric. Therefore, specific splitting strategies, taking the numeric variable into account, have to be used. This section gives an overview of the most famous splitting criteria for the regression task.

Standard Deviation Reduction of the Target Variable This criterion [21] follows the idea of the information gain: an increased purity of the target variable is wanted. For that, the variance of the target variable should be maximally reduced by the split. This is done by choosing the split resulting in the highest standard deviation reduction (SDR) given in Equation 3.9.

$$SDR = sd(T) - \sum_i \frac{|T_i|}{|T|} sd(T_i) \tag{3.9}$$

$$sd(T) = \sqrt{\frac{1}{N}(\sum_{i=1}^{N} y_i^2 - \frac{1}{N}(\sum_{i=1}^{N} y_i)^2)} \tag{3.10}$$

Here, T is the set of examples reaching this leaf node and $T_1, ..., T_n$ are the sets resulting by the split. An advantage of this approach is its fast calculation. A disadvantage is that the quality of linear models are quite independent of the variance of the dataset they are built on [78, 96]. Basing the splitting decision on the SDR reduction does not necessarily imply an increased quality of the new linear models or that the best splitting points are found. Suppose a piecewise linear function a model tree would be able to fit by finding the correct splitting points separating the pieces. Using the highest SDR as splitting decision will be fast, but seldom result in finding the splitting points separating the linear functions. Consequently, the fit of the linear models in the created subspaces will not be optimal.

Prediction Error Reduction Another approach to choose the best splitting point is to directly evaluate the performance of the models in the newly created leaves after the split and to compare them with the current performance [78]. If an improvement, i.e. a decrease of the combined model error, is observed, a split should be performed. In comparison to the above mentioned standard deviation reduction of the target variable, the splitting decision is here calculated by including the actual reduction of the prediction error. The calculation of the error depends on the actual application (see Table 2.4 in Section 2.2.1 for possible error estimates) as well as the error combination approach of the child nodes.
A popular approach is the mean squared error reduction, which is shown in

Equation 3.11.

$$MSER = MSE(T) - \sum_i \frac{|T_i|}{|T|} MSE(T_i) \tag{3.11}$$

Instead of just focusing on the target variable, the outcome of the prediction is considered in this measure. The mean squared error (cf. Equation 3.12) is compared for the examples for and after the split and further weighted by the example distribution to the subnodes (cf. Equation 3.11).

$$MSE(T) = \frac{1}{|T|} \sum_i^{|T|} (y_i - \hat{y}_i)^2 \tag{3.12}$$

The splitting decision resulting in the highest error reduction is chosen. The squared error is also often called the *residual sum of squares (RSS).*

Residual Distribution Test on Mean and Variance This splitting criterion [25] is based on the residuals received from applying a linear model, which is fitted on the examples in the leaf node. Two subsamples of positive and negative residuals are formed and compared for each regressor. The subsamples should not be very different if the linear model fits the examples. Otherwise, a split should be preferable. The difference test is based on the differences in sample mean and sample variance. For example, with a convex regression function in the leaf node, a split should be beneficial. This would be detected by a larger variance in the positive residual group compared to the negative residual group.
Let I_r be the number of examples with a positive residual (I_1) or a negative residual (I_2). Furthermore, let $\bar{x}_{.jr}$ and s^2_{jr} be the mean and the variance of all examples with residual class r in attribute X_j. $s^2_{j.}$ is the pooled variance estimate. The distance of each attribute value from the attribute mean in class r is defined as $d_{ijr} = |x_{ijr} - \bar{x}_{.jr}|$. $\bar{d}_{.jr}$ and v^2_{jr} are the mean and the variance of the difference values d. Formula 3.13 shows the mean test and Formula 3.14 the variance test (Levene test) respectively.

$$MT_j = \frac{(\bar{x}_{.j1} - \bar{x}_{.j2})}{s_{j.}\sqrt{I_1^{-1} + I_2^{-1}}} \tag{3.13}$$

$$VT_j = \frac{(\bar{d}_{.j1} - \bar{d}_{.j2})}{v_{j.}\sqrt{I_1^{-1} + I_2^{-1}}} \tag{3.14}$$

The variable X_j with the smallest p-value associated with the two statistics (Student‘s t-distribution with $(I_1 + I_2 - 2)$ degrees of freedom) is chosen as the splitting attribute and the splitting value is the average of the two sample means ($\bar{x}_{.j1}$, $\bar{x}_{.j2}$).

Residual Distribution Test with Pearson Chi-Square Test Two drawbacks of the residual distribution test on mean and variance is that it cannot be performed on categorical attributes and pairwise attribute interactions are not considered. Both drawbacks are eliminated with the following approach [92]:

1. A linear model is fitted on the examples in the leaf node and two subgroups with positive and negative residuals are formed.
2. Curvature test on each numerical-valued variable: The variable range is divided in four groups at the sample quartiles (other divisions are imaginable). A 2 x 4 contingency table is constructed with the residual groups as rows and the range groups as columns. The entries are the numbers of occurrences in the example data. The Pearson chi-square test is performed on the table and the p-value is received.
3. Curvature test on each categorical variable: This time, the contingency table is formed by using the different categories of the categorical variable as columns. Table columns with a sum of zero are omitted. Then the Person chi-square test is applied and a p-value is received.
4. Interaction test on each pair of numerical-valued variables: The space spanned by the two variables is divided into four quadrants by splitting each variable range into two halves at the sample median. Again, a 2 x 4 contingency is built with the residual groups as rows and each quadrant as a column. The chi-square test is performed and the p-value is computed. Zero columns are again omitted.
5. Interaction test on each pair of categorical values: The chi-square test is based in this setting on the contingency table with columns representing all attribute category combinations between the two attributes.
6. Interaction test on each pair of categorical and numerical-valued variable: The attribute space of the numerical-valued variable is divided into two subspaces at the sample median and the categorical variable in the number of categories. The combination of all subspaces of the two variables are used again as the columns in the contingency table, on which the chi-square test is performed and the p-value is calculated.

If the smallest p-value of all tests is from a curvature test, this variable is finally chosen as the splitting attribute. If it is based on an interaction test, both variables are potential splitting attributes. If both attributes are numerical-valued, the attribute resulting in the split with the smallest total SSE over all models in the subspaces is used. For this evaluation a split at the sample mean is supposed. If at least one attribute is categorical, the one with the smaller curvature p-value is chosen as splitting attribute. Once the splitting attribute is chosen, the splitting value is needed to form a valid splitting point. The splitting value is either found by choosing the sample median of the attribute, or by performing a greedy search to find the value that minimizes the total SSE. Both approaches are possible, while the greedy approach is much more computationally expensive.

EM Splitting Approach For this splitting criterion [33], the best split point for the regression task is again found by a transformation in a classification problem. The EM algorithm is applied to all examples in the leaf node and two Gaussian clusters are formed. Then each example is labeled with class 1 or class 2, depending on the assignment probability. The corresponding split point (split attribute and split value) is found by evaluating each separation with a goodness of split measure used for classification trees. An exemplary measure is the Gini gain and the splitting decision resulting in the highest gain is chosen.

3.2.5 Pruning of Decision Trees

Mainly two approaches for pruning exist: pre- and postpruning, which can also be combined. While prepruning hinders the tree growth in the induction process, postpruning is applied to the final tree to prune harmful subtrees. Both approaches are explained in the following. Although pruning improves the prediction accuracy in many cases it comes along with three drawbacks. First, most pruning techniques have an impractical time complexity and second, pruning is usually a greedy strategy and thus cannot guarantee globally optimal trees. Additionally, some studies show that pruning in the batch setting only works for data sets with simple concepts. If more complex concepts have to be induced, it is not necessarily better than the normal top-down induction approach [39].

Prepruning

Prepruning is applied during the tree induction process and hinders leaf nodes to be further split if the benefit is not observable. This avoids unnecessary work in the tree construction phase [20] and is usually done by testing a stopping criterion based on a significance test. One example is to compute a statistic indicating if a node will be pruned away in the subsequent pruning phase [111]. Expanding this node in the building phase is thus useless and can be avoided. Another straightforward method is to use a user-defined threshold like a maximal tree depth or a minimal value for the splitting criterion (e.g. gain ratio). Finding the proper threshold is a major challenge in prepruning. In the online setting, the stopping test is further supported by a statistical test stating if enough examples are included in

the stopping test for a valid decision.
A drawback of this approach is that it uses only limited information for the stopping criterion. It is hard to forecast how the subtree will evolve and if the split would be beneficial if the whole subtree will be created. Thus, the approach of first creating the whole decision tree and then to prune it (postpruning) is often preferred over prepruning.

Postpruning

In comparison to prepruning, postpruning is applied to the final decision tree, which probably over-fits the training data. Subtrees were built too specific on anomalies of the training set, resulting in a decreased prediction performance. Consequently, these subtrees, which are not considered as significant or reliable, have to be pruned from the tree to keep the performance as well as the processing speed high. Postpruning can be applied in a top-down or bottom-up fashion. Top-down pruning starts with the root node and on the path to the leaf nodes, each subtree is tested for its contribution to a criterion, which is mostly the prediction accuracy. If the subtree is considered as harmful for the criterion, it is completely pruned from the tree although good and contributing subtrees may be included. Bottom-up pruning, on the other hand, starts at the leaf nodes in the tree and tests level by level to the root node each subtree, belonging to the current node, for its contribution to the criterion. Consequently, weak subtrees are pruned right away and consequent subtree tests on higher levels are already performed on the improved replacement. Thus, in comparison to the top-down approach, trees are pruned more specifically, removing only the harmful regions of the tree. In the following some postpruning approaches are further explained.

- **Cost-complexity pruning** [21]: From the final decision tree, $k-1$ smaller trees are created by repeatedly removing one or more subtrees with the lowest increase in the apparent error rate per pruned leaf. From the k trees, the best pruned tree with the lowest generalization error is then chosen as the final tree. To evaluate the error rate, a pruning set should be used if the dataset is large enough to be split in training and pruning examples.

- **Reduced error pruning** [108] traverses the final tree from the bottom to the root. Each internal node is tested if a replacement (including its subtree) with a leaf node does not reduce the trees accuracy. If so, the subtree is pruned from the tree. This is continued until further pruning is harmful. Again, a pruning set should be used for accuracy estimation.

- **Minimum error pruning** [103] performs a bottom-up traversal of the decision tree. Each subtree is tested for pruning by using the l-probability error rate estimation, a correction of the simple probability estimation using frequencies. Pruning is accepted if the error rate is not increased.

- **Pessimistic pruning** [107] avoids the need for a separate pruning set, as the error estimation is based on the training set. This comes with the disadvantage of a too optimistic and, therefore, strongly biased error estimation. To counter the bias, the error estimates are adjusted by adding a penalty.
- **Error-based pruning** [110] is an improvement of pessimistic pruning and uses a far more pessimistic estimate. It is integrated in the C4.5 algorithm. This procedure performs bottom-up traversal and for each internal node the number of errors in its subleaves are predicted. This is done by multiplying the number of examples at each leaf with an upper limit of a probability confidence limit of an error in this leaf (calculated by using the binomial theorem). Additionally, the number of errors if the node was a leaf are also calculated. The leaves are finally pruned if the number of predicted errors after pruning is less than the sum of predicted errors across the leaves.
- **Optimal pruning** [16, 6] tries to find the optimal pruning by using dynamic programming. Algorithms guaranteeing optimality are OPT and its improvement OPT-2.
- A pruning strategy based on the **MDL principle** [98] can also be applied for pruning. Again, this approach calculates the coding size of the decision tree and its error descriptions. The (pruned) tree having the optimal MDL is then taken as the final result.

Several studies have been performed to compare the different pruning techniques [108, 99, 41]. Results show that some approaches (e.g. cost-complexity pruning, reduced error pruning) have a bias towards over-pruning, and others (e.g. error-based pruning, pessimistic error pruning and minimum error pruning) tend to under-pruning. Additionally most studies stated that there is no single pruning method outperforming all others under all circumstances. Another study [98] concluded that pruning based on the MDL principle seems to outperform other pruning strategies, as it comes to more accurate results and smaller trees.
In the online setting, over-fitting correction is only one application field of pruning. Another one is the correction of wrong splitting decisions, which were made based on insufficient data at the time of the split. As data is constantly arriving, splits could be made too early, whereas later decisions would have led to another split decision. Another application for postpruning in the online setting is the tree correction due to detected concept drift [70]. As mentioned in Section 2.3, concept drift outdates specific parts of the decision tree, leading to a decrease prediction performance. Pruning can

remove or replace these parts of the decision tree. While the aforementioned postpruning methods are developed for the batch setting, their application to the online setting is possible as well, but has to be done with caution. Postpruning is normally done after the decision tree has been learned on the complete training set. In the online setting, the training set could be infinite. Consequently, there will be no final tree to prune. Tree pruning in this setting has to be done periodically. After a specific number of examples or after a special event (e.g. detected concept drift), the tree could be tested for pruning. This could be, e.g., after each example has traversed the tree [106]. Testing after each example is very time consuming and can result in decision tree algorithms improper for fast data stream learning (see Chapter 5). Decreasing the testing frequency seems appropriate, but raises the risk of delayed splitting error corrections.

3.2.6 The Lookahead Strategy

One problem in decision tree induction is the greedy nature of the algorithms. Split decisions are usually based on the current structure of the tree and can result in locally optimal trees. One step towards globally optimal trees is to use lookahead strategies. There, splits are not only based on the current tree but also on forecasts reflecting the splitting outcome of each possible split decision. Forecasts can be approximations of the outcome of the split or exact tests of the outcomes (e.g., the expected quality of the models after the split). The first approach is called approximate lookahead and the last precise lookahead in this work. Let us reuse the regression task from the party example (see Figure 3.1). An optimal algorithm to predict the amount of people liking the party would be to use a model tree. This tree could use a linear regression model in each leaf. To optimally match the concept, splits are needed at 100 and 200 party guests. Now let us suppose we have an approximate lookahead approach in the splitting decision, e.g. the variance reduction of the target variable. And let us further suppose it would result in a split at 50 or 250 guests. Consequently, a suboptimal split is chosen. By using a precise lookahead, e.g. by applying linear regression models in the next level introduced by the split, the correct split points at 100 and 200 guests would be found. While resulting in good splitting decisions, precise lookahead is expensive in the perspective of memory consumption and update time. Both can raise problems in the online setting. Approximate lookaheads are fast, needing less memory, but resulting in suboptimal splitting decisions.
The party example is an easy regression task example. The correct split points can be found by making a forecast over the next split level. Real world problems are mostly very complex and lookaheads over several splitting levels could be useful. The more forecast levels are considered, the closer the algorithm is to an exhaustive search. Optimally, the level is cho-

sen depending on the resources available. Consider the case when decision trees are used in a streaming setting: depending on the frequency of arriving examples, the learner has more or less time to find an optimal splitting decision. If more time is available, the lookahead can be expanded over more levels and if time is limited, less levels are used for the lookahead. Consequently, the best splitting decision can be found using the currently available resources [40]. However, no agreement on the usefulness of lookaheads over several levels can be found. Some surveys claim an improvement, while others state that this method may even be harmful.

3.3 Decision Tree Algorithms

One of the first publications on decision trees was E. B. Hunt's work on *concept learning systems* [68] in 1966. It served as a basis for some of the more complex algorithms as, e.g., the ID3 algorithm [107] developed by Quinlan *et al.* in 1986, or its famous successor C4.5 [110]. Around the same time, in 1984, L. Breiman *et al.* independently published the book *Classification and Regression Trees* (CART). It described the generation of binary decision trees for the classification as well as for the regression task. The next step in the evolution of decision trees was to not only perform predictions in the tree leaves based on single values (e.g., the mean value of the target value), but on more or less complex prediction algorithms. This subgroup of decision trees is called *model trees* [96].
This section introduces the most well known decision tree algorithms. In the following, they are presented by their field of application: Decision trees learned to predict a numeric variable are explained in the regression task sections and algorithms to predict a categorical target variable are shown in the classification task sections. Furthermore, the algorithms are divided into batch and incremental learners.

3.3.1 Classification Trees

Batch Algorithms

Batch classification tree algorithms can be divided based on their mechanism in three different groups (based on Suknovic *et al.* [127]):

- C4.5-like
- Fact-like
- Others

All C4.5 like algorithms follow the basics steps of decision tree induction that are explained in Section 3.2.2:

1. Generate split candidates of dataset D.
2. Evaluate splits with a specific measure to identify the best splitting decision dividing D into subsets $D_1, ..., D_n$.
3. repeat steps 1. and 2. recursively until a stopping criterion is reached (e.g, maximal tree depth or purity).

Famous algorithms in this algorithm group are, e.g., ID3 [107], C4.5 [110] and CHAID [79]. ID3 (Dichotomiser) uses the information gain criterion (cf. Section 3.2.4) to choose the best splitting decision. It deals with noisy and incomplete data and is consequently a robust algorithm. The first version of the algorithm works only on nominal attributes. Many extensions are know as, e.g., ID3-IV, GID3 and GID3*. A very interesting variant is LSID3 [40]. This type of anytime decision tree uses an advanced lookahead strategy to decide for the next split. Depending on the resources available, the lookahead may span over different depths of the tree. The more time is available, the more can be used in the lookahead phase.
C4.5 [110] is a successor of ID3 and is currently considered as state of the art. There is also a commercial version (C5.0) available, which implements some quality improvements over the freely available version. Additionally, it is much faster and produces smaller trees of equal quality compared to C4.5. C4.5 can now also deal with numeric attributes and still handles missing and noisy data. It uses the gain ratio as splitting criterion and applies pessimistic pruning (cf. Section 3.2.5). An often used implementation of the C4.5 algorithm is called J48 and can be found in the Weka software.
CHAID [79] is another C4.5-like algorithm. It uses numeric or nominal variables as input, but in the case of a numeric split it tries to find splits having approximately the same amount of variables in each group. Missing data is handled as a separate value and significant splits are based on chi-square tests.

Fact-like induction algorithms also apply a divide and conquer manner for the decision tree learning. However, they work not exactly in the same way as C4.5-like approaches. The following list briefly gives the main induction steps:

1. To find the best splitting attribute X_j all remaining attributes are evaluated regarding the target variable y. A new node in the tree is formed with this attribute.
2. Then, the branches of the tree are created by identifying the most appropriate split boundaries of X_j.
3. Like C4.5-like algorithms the two steps are repeated until a stopping criterion is reached.

Exemplary algorithms in this group are FACT [94], QUEST [93] and CRUISE [81].
The basic idea of FACT [94] is to combine the potential of CART with the power of linear discriminant analysis, i.e. an algorithm that uses the speed of the linear calculations and outputs an interpretable tree. Therefore, FACT does not combine splitting variable selection and splitting value selection, but treats these steps individually. While the F-statistic is used to find the splitting attribute, the splitting value is found in a next step by applying a linear discriminant analysis. FACT is also able to induce linear combination splits. Therefore it actually applies a recursive linear discriminant analysis on all the variables. It also uses a specific stopping criterion, i.e. a set of rules, instead of learning the complete tree and then to prune it. The stopping criterion applies if the error does not decrease anymore or if the number of examples in a leaf is too small.
QUEST [93] is similar to FACT, but tries to get rid of the variable selection bias towards nominal attributes. It also applies univariate splits as well as linear combination splits. The best splitting attribute is found by applying a statistical test (ANOVA F-test, Levene's test or Pearson's chi-square) depending on the attribute type. Additionally, Quadratic Discriminant Analysis (QDA) is used to identify the best splitting value for numeric attributes. It manages to overcome the variable selection bias problem of FACT and results in binary splits only. Moreover, it may apply pruning or a direct stopping criterion.
CRUISE [81] also tries to reduce the variable selection bias and furthermore tries to create trees which are better to understand. To minimize the selection bias it again applies statistical measures to identify the most significant variable and moreover, handles different numbers of missing values by a bootstrap bias correction. Multiway splits are created by univariate splits as well as linear combination split. This results in very compact trees that may be much easier to interpret than binary trees and are learned faster. As a specialty, it furthermore incorporates local interactions of variables, which also has an effect on the tree depth. It was shown on an extensive study that the quality of the resulting trees is comparable to other classification tree methods, but that Cruise is significantly faster. This was also true in the presence of missing data.

Other algorithms worth mentioning are O-Btree [42] and Public [112].
O-Btree uses the C-SEP attribute selection measure (cf. Section 3.2.4). It induces a strictly binary tree (B-Tree) using the ORT distance measure. On synthetic data sets it succeeds in finding the optimal tree producing no errors and moreover, finds minimal trees on real world data sets producing the least errors.
PUBLIC is a classification tree induction method that uses the MDL principle for the growing of the tree and the pruning. More specifically it uses prepruning. A binary tree is created, where each node contains a sorted list

for each attribute and a class histogram. The splitting criterion is based on the class distribution of the resulting partitions (S_1, S_2) and measured by the Entropy (cf. Section 3.2.4). Discrete and continuous variables can be used for this induction method.

Recently, a framework that allows to combine specific parts of all algorithms was presented [127]. This also allows creating completely new induction methods, that may outperform former ones under specific conditions.

Incremental Algorithms

Incremental algorithms emerged when datasets became too large to be stored in main memory. Consequently, the usually used batch algorithms were not able to process the whole dataset anymore [29]. The first field of active research in the area of incremental tree induction was the induction process of incremental classification trees. First approaches are based on the idea to only update the existing classification tree with the newly available data, instead of a resource consuming recalculation. Examples of this approach are the incremental versions of the ID3 algorithm. There are several algorithms resulting from a continuous improvement process: ID3' [119], ID4 [119], ID5 [131], ID5R [132], and ITI [133]. Each approach learns an initial tree, which can be updated with new examples, instead of recalculating the whole tree. With every new arriving example, the whole tree is tested if its structure is still valid for the new data basis, i.e. all examples seen so far. This is done by testing the tree top-down if each splitting decision is still the best choice. A restructuring process starts, if another splitting decision would be chosen on the new data basis. Depending on the algorithm, this restructuring ranges from subtree deletions, complete subtree recalculations or more advanced restructurings. To make the restructuring process possible, most of the algorithms have to store all so far seen examples. Consequently, this kind of algorithms is resource consuming and less intended to be applied to data streams or any application field where fast adaptation is needed.
Another algorithm family, especially useful on data streams, are the Hoeffding trees [35]. These algorithms are the most successful incremental decision tree algorithms and famous representatives are VFDT [35], CVFDT [67], VFDTc [47], and iOVFDT [143]. In comparison to the aforementioned algorithms, they use a statistical measure to find a valid splitting decision calculated on only a fraction of the available data. To decide how many examples are needed to make this splitting decision valid, the Hoeffding bound is used [66]. Consider a random variable r with range R (e.g. a range of 1 for a probability) and n observations r_n. The computed mean over these observation is $\bar{r}$. The Hoeffding bound now states that with probability 1- δ the true mean $\bar{r}_{true}$ for the variable r is at least $\bar{r} - \epsilon$, where $\bar{r} = \frac{1}{n}\sum_{i=1}^{n} r_i$

and ϵ is the value of the Hoeffding bound:

$$\epsilon = \sqrt{\frac{R^2 ln(1/\delta)}{2n}} \tag{3.15}$$

The splitting decision is still based on a splitting criterion $C()$ (e.g. information gain or Gini gain). Let SD_1 be the splitting decision with the highest splitting criterion $\bar{C}(SD_1)$ after seeing n examples. SD_2 is the splitting decision with the second highest splitting criterion. The Hoeffding bound is now used to judge if it is valid to choose SD_1 as the best splitting decision, i.e. SD_1 is the best choice after seeing n examples and will still be the best choice after seeing an infinite number of examples. $\Delta\bar{C} = \bar{C}(SD_1) - \bar{C}(SD_2) \geq 0$ is the difference between the splitting criterion of the current best and second-best choice. The Hoeffding bound now guarantees for a desired δ, that SD_1 is the correct choice with a probability of $1 - \delta$ if $\Delta\bar{C} > \epsilon$. The incorporation of Hoeffding bounds guarantees an upper error bound compared to the result of a conventional batch learner. That means that the results of the Hoeffding tree and a conventional (batch) learner are asymptotically the same. An advantage of the Hoeffding bound is that it is independent from the probability distribution generating the observations. A disadvantage is that it is more conservative than distribution dependent bounds and consequently, needs more examples to reach the same δ and ϵ. Recent work suggests that the usage of the Hoeffding bound may not be appropriate in every case, but rather McDiarmid's bound [117].
The VFDT (Very Fast Decision Tree learner) algorithm is a well-known implementation of a Hoeffding tree. It was further improved in speed and received an advanced memory management system. Speed was mainly increased by not testing the leaf node after each example for a split, but only after N_{min} examples. Furthermore, the tie-breaking rule is introduced. Two equally good split points in a Hoeffding bound need too many examples and time to decide for a splitting decision (if there will ever be a decision). The tie is solved and the current best splitting decision is chosen if $\Delta\bar{C} < \epsilon < \tau$, where τ is a user defined threshold. VFDT stores no examples. To perform the splits, only abstract statistics are stored and consequently, VFDT is very memory efficient. Nevertheless, if a memory limit was introduced and when it is reached by the model, VFDT deactivates the least promising leaves to give the promising leaves a little more time to develop. Additionally, poor attributes can be dropped from the splitting decision if their splitting criterion difference to the current best splitting decision is greater than ϵ. This also frees additional memory. For splitting decisions, the information gain or Gini gain can be chosen. The prediction is performed by choosing the dominant class in the leaf.
VFDT cannot handle concept drift. Consequently, it has been extended to CVFDT and VFDTc, which can detect concept drift and replace subtrees with newly, on the new concept learned, subtrees. A very recent extension

of VFDT is ioVFDT (Incrementally Optimized Very Fast Decision Trees) [144]. Its main improvements consider its application on high-speed data streams, an automatic identification of the parameters for the growth of the tree and the creation of balanced models concerning accuracy, tree size and induction time. All these requirements are formulated and solved as an optimization problem. In the leaves, a weighted Naive Bayes approach is installed to return a prediction value. The resulting new type of nodes is called functional tree leaf. In a bunch of experiments ioVFDT nearly reaches the accuracy of C4.5 and also results in smaller models with even higher accuracy compared to other algorithms [143]. This algorithm is also available in the MOA framework.

3.3.2 Regression Trees

Decision trees for the regression task are a relatively young field of research. The prediction of a continuous variable was in focus after developing algorithms for the classification task, which seems to be easier. Consequently, the classification task is more developed than the regression task. Nevertheless, also in this field of research, a large number of algorithms are available. Main research was done in the lookahead procedure to optimize the interaction of the splitting method and the prediction method. This is shown in the following by first explaining the most important batch algorithms. After that, an overview of the incremental regression tree algorithms is given. Due to the fact that only the regression task and especially the incremental algorithms are in focus of the next chapters, a more detailed overview is given for these algorithms.

Batch Algorithms

Due to the huge amount of available batch regression trees, only the most important ones are presented in the following. These algorithms are often used as baseline algorithms for improvements and incremental adaptations, which will be shown in the incremental algorithm section.

CART Leo Breiman, Jerome H. Friedman, Richard A. Olshen and Charles J. Stone published in 1984 the book Classification and Regression Trees (CART [21]). It describes the induction of binary decision trees. Classification, as well as regression trees. The decision tree community is strongly influenced by this work and many regression tree algorithms are based on the ideas mentioned in the work. Due to the fact that Breiman showed a common induction process for a classification tree and a regression tree, the splitting criterion focuses on the same goal: the reduction of impurity. While the Gini gain is used for the classification tree, the standard deviation reduction of the target variable was used for the regression tree. The splitting decision showing the highest standard deviation reduction is used. Instead of a model, a constant value, the mean of the target value of all examples in the node, is fitted in each leaf node and used for prediction. To achieve better results, CART uses the cost complexity pruning approach.

M5 and M5' The M5 algorithm was developed 1992 by Quinlan [109]. It was one of the first algorithms creating a model tree to solve the regression task with piecewise linear functions. So far, regression trees performed the prediction in each leaf node by returning the mean value of the target variable (e.g. CART). Model trees create more compact and consequently, more readable trees. M5 creates and improves the tree by using the standard deviation reduction (SDR, cf. Section 3.2.4) splitting criterion. The induction process is stopped, if the variance of the target variable is already small or too few examples are available in this leaf node for further splitting. Last, a bottom-up pruning is performed by comparing the estimated error of each internal node's model with the errors of the models in its subtree. If the error of the internal node's model is lower, the subtree is pruned from the tree. For the prediction task, multivariate linear models are learned in each node (internal nodes and leaf nodes). Instead of using all attributes of the dataset in the regression task, only a selection of attributes is considered as useful. These attributes are the ones included in splitting decisions or in models below this node (in its subtree). Additionally, the models are further simplified by removing attributes having such a small effect, that they are increasing the estimated error. To improve the prediction accuracy, an approach called smoothing is applied. After traversing the tree, the model in the leaf node is used to give a prediction for the examples. The predicted value at the leaf node is then further adjusted by the predictions of the internal node models on the path to the root node on the same example.
The M5' algorithm was developed 1997 by Wang and Witten [140]. Corresponding to the authors, it is a "rational reconstruction"of M5. Consequently, the base algorithm is identical to M5 and some recommendations for unclear steps in M5 are made. For example, it is proposed that attributes, that were removed from the model due to their effect on increasing the estimated error, should still be included in higher level models. Additionally, minor changes are made to further improve the algorithm: Two methods from the CART system have been adopted to handle enumerated variables and missing values.
As a drawback of M5 and M5', one can discuss that the lookahead strategy of the splitting decision is not coordinated with the prediction approach in the leaves. While using multivariate linear models for prediction, the splitting decision is only based on the standard deviation reduction of the target variable and consequently, is only an approximate lookahead approach.

HTL Decision trees are very user friendly, as they can be well interpreted and information can be extracted from their structure. Nevertheless, using the same model creation process in each subspace could be a limitation to the prediction accuracy. A higher accuracy could be reached by using the best fitting approach in each subspace. That is the basic idea behind the hybrid system HTL [129]. HTL builds a binary model tree with the best fitting model type in each subspace (leaf node). These model types are from a collection of available models as, e.g., linear or kernel based functions.
The tree induction process follows the MSE reduction principle (see Section 3.2.4). But instead of using the actual model predictions $\hat{y}$, the average of the target variable ($\bar{y}$) is used as prediction in the MSE calculation. Categorical as well as numerical attributes are included in the induction process. Tree growth is stopped if either there are not enough examples in the leaf or if the statistic *Coefficient of Variation* of the target variable y is below a specific threshold. Pruning is done by using a pessimistic heuristic estimation of the true node error on the training set. Subtrees showing no error reduction are removed. In the leaves, HTL is now able to fit several alternative regression models to improve the prediction accuracy. While the accuracy is raised in this approach by using the optimal model in the subspace, the splitting decision is still not based on the actual model accuracy and, again, only an approximate lookahead approach is used.

TREED REGRESSION Another early model tree induction algorithm is TREED REGRESSION [5]. There, a binary decision tree is induced holding simple linear regression models in the leaves. The focus of this approach lies in the splitting decision: In order to decide for a splitting decision, simple linear regressions are calculated for each subspace. Each independent attribute is used as the regressor in the model and the best linear model is determined for each subspace independently. The splitting decision with the best model performances is used as the final splitting decision and the, already learned, models are used in the leaf nodes. To find the best splitting decision, many models have to be fit, which leads to a performance loss.

RETIS RETIS [78] is another model tree induction algorithm with multivariate linear regression models in the leaf nodes. In comparison to the afore mentioned algorithms, RETIS bases its split decisions on the actual error reduction, induced by linear models created by the split. To make this happen, two models (for a binary split) have to be calculated for each possible splitting decision and for each leaf node. The MSE reduction is calculated on the actual predictions of the models and the splitting decision resulting in the maximal MSE reduction is finally chosen. By using this kind of precise lookahead, high resources concerning memory and calculation time are needed. This is the main point of criticism of this approach. The system hardly scales up to large datasets, especially in the presence of many continuous variables. Consequently, efficient methods were created to build decision trees similar to the ones created by RETIS without the computational drawbacks [130, 137]. To keep the tree small, reduced error pruning is used.

MAUVE Due to performance issues, approximate lookaheads as, e.g., the variance reduction of the target variable, are preferred over a precise lookahead as used by RETIS. Although choosing an approximate lookahead may not have huge impact on the prediction accuracy, the tree structure will be misleading and huge decision trees will be created. From this point of view, the MAUVE (M5' Adapted to use Uni-VariatE regression [135]) algorithm has been developed, which uses a little more complex approximate lookahead. It is approximately as efficient as the variance based approach and results in more explainable and compact trees. MAUVE is based on the M5' approach with changes in the applied splitting heuristic, the stopping criterion and the pruning approach. For each numeric attribute and each possible splitting decision for this attribute, a simple linear regression to the target variable is calculated for each resulting subspace. In contrast to TREED REGRESSION, only the splitting variable is used as regressor in the simple regression. The split showing the highest standard deviation reduction of the residuals is then chosen as the best splitting decision. For categorical variables, the split decision follows the suggestions of M5'. Although simple linear regressions are used to identify the splitting decisions, multiple linear regressions are used in the leaf nodes for predictions. In contrast to RETIS, not all variables are included as regressors. As all splitting variables on the path to the leaf show a linear correlation to the target attribute, they are included in the model. A variable selection process further evaluates the remaining attributes whether to include them in the model of the leaf. If there are not enough examples in a leaf, i.e. at least two times the number of attributes, the node is prevented from splitting. A subtree is pruned away if its error is larger than the error of a model in its root node. Experiments showed that MAUVE comes to better (smaller but equally accurate) models compared to M5', RETIS or TREED REGRESSION.

SUPPORT The afore mentioned algorithms base their split decisions either on variance reduction or on forecasts to see how the regression models may perform in the subspaces. Another group of algorithms also uses multiple linear regression models for predictions in their leaves, but their splitting decision is based on a classification setting. One of these algorithms is SUPPORT (Smoothed and Unsmoothed Piecewise-Polynomial Regression Trees [25]), a nonparametric algorithm for binary model tree induction. The used splitting criterion is the residual distribution test on mean and variance (cf. Section 3.2.4), which evaluates the distribution of the positive and negative residuals of the fitted model in the leaf node. Categorical attributes cannot be taken into account. For prediction, the algorithm uses polynomial functions as models in the leaves. If a smoothed prediction is wanted, these polynomial models are combined in a weighted average approach. SUPPORT uses a prepruning approach during tree induction: a cross-validation multi-step lookahead strategy to accurately stop the tree growth.

GUIDE The GUIDE algorithm (Generalized, Unbiased Interaction, Detection and Estimation [92]) was designed to reduce the variable selection bias present in many algorithms, as e.g., in the CART algorithm. To accelerate the splitting decision, GUIDE also transforms the problem in a classification setting. In contrast to SUPPORT, it uses the residual distribution test with the Pearson chi-square test (cf. Section 3.2.4) to be able to include categorical attributes in the splitting decision and to consider pairwise attribute interactions. Furthermore, the selection bias is strongly reduced by a bootstrap approach. The user can choose one of three roles for each ordered predictor variable: split selection only, regression modeling only, or both. This choice has further impact on the splitting decision, the tree structure and the tree performance. Pruning is done by a cost-complexity pruning approach using an independent test set or, in its absence, by cross-validation. In more detail, first the complete tree is induced and then pruned back until only the root node is left. This leads to a sequence of nested trees for which the prediction mean squared error (PMSE) is calculated by a cross-validation. The tree with the smallest PMSE is finally chosen. Multiple linear regression models are used in the leaves for prediction.

SECRET One of the main challenges of batch algorithms is to handle large datasets. SECRET (Scalable EM and Classification based Regression Trees [33]) is a model tree which is accurate and truly scalable for large datasets. As GUIDE and SUPPORT, it transforms the splitting task into a classification problem to boost the induction process. The best splitting decision is found by applying the EM splitting approach (cf. Section 3.2.4). The calculation is further boosted by using scalable versions of the EM algorithm for Gaussian mixtures [19] and tree construction [58]. Predictions are achieved by least square linear regression in the leaves. A pruning approach is not explicitly mentioned by the authors. Evaluations showed comparable results to other regression tree algorithms (e.g. GUIDE), but less computation time for large datasets.

Incremental Algorithms

The domain of incremental regression tree algorithms is, in comparison to the classification tree domain, an almost neglected field of research. Only a small number of algorithms has been developed so far. Nevertheless, due to the new era of Big Data, this group of algorithms gains importance. All of the existing algorithms are more or less based on a batch algorithm shown in the section afore. Adaptions were made to adjust the algorithms to the new needs in online learning.

IMTI-RD IMTI-RD is an incremental adaptation of RETIS. The properties of RETIS described in Section 3.3.2 hold unless corrected in the following. Each split candidate is evaluated by the actual error reduction performed by this split, and the one with the highest reduction is chosen. To do so, *spst* holds a linear model m_{ikL} for the left subspace created by the split on attribute i with value k, and m_{ikR} for the right subspace model. Model m as well as all submodels m_{ikL} and m_{ikR} are updated with each example arriving in this leaf and the residual sum of squares error (RSS) is accumulated. As storing these submodels for all values of an attribute is memory and time consuming, only κ potential splits are stored for each attribute. An adapted Chow test is used to find the best possible split. This statistic is distributed according to Fisher‘s $\mathcal{F}$ distribution and the candidate split with the smallest p-value smaller than α_{split} is chosen. Wrong splitting decisions can be corrected after each example by checking each node on the path from the root to the leaf for pruning. In each node, the RSS made by m is compared to the RSS made in all leaves of the nodes' subtree. The used test statistic is again $\mathcal{F}$ distributed, and the entire subtree is pruned when the calculated p-value is less than α_{prune}.

IMTI-RA IMTI-RA is the incremental extension of SUPPORT (cf. Section 3.3.2), and consequently, the splitting decision is based on the distribution difference of the regressor values from the two sub-samples associated with the positive and negative residuals. If the approximated function is almost linear in this node, the positive and negative residuals are distributed evenly. This is tested by two statistics evaluating the differences in mean and variance for each regressor. Both statistics are assumed to be distributed according to Student‘s t distribution. If the null hypothesis (function is linear) can be rejected with a confidence of $1 - \alpha_{split}$, the leaf node is split. Consequently, the split statistic *spst* has to store the mean and variance values for the positive and negative residual groups for each attribute. Additionally, *IMTI-RA* uses a pruning method to correct erroneous splits. With each example, each node on the path from the root to the leaf is checked if the null hypothesis used for splitting cannot be rejected anymore with confidence $1 - \alpha_{prune}$. If so, the node with its subtree is pruned from the tree.

FIMT Another adaptation of a batch algorithm to the online setting is FIMT (Fast and Incremental Model Tree) [69]. It is based on the principle of the M5 algorithm and inspired by the VFDT algorithm. It was implemented in the VFML library of Domingos and consequently many features of the VFDT algorithm were adopted (e.g., statistical bound, tie breaking, periodical split test, memory management). It is an efficient and fast algorithm which needs no example storing. Each arriving example is stored only once and then discarded. Memory is only occupied by the splitting statistics *spst* and the models. The splitting decision of FIMT is identical to the one of the M5 algorithm: the maximal standard deviation reduction (SDR) of the target variable through the possible split (cf. Section 3.2.4). As the training examples are not available at once in the online setting and no examples themselves are stored, *spst* has to save the necessary information to enable the SDR calculation. This is done by storing in the splitting statistic *spst* for each leaf node an extended binary search tree (E-BST) for every numerical attribute. Each value of the numerical attribute is a node in the binary tree. Additionally, six properties are attached to each node which can be easily incrementally updated. These values are needed to efficiently calculate the SDR for each value of each attribute (each possible split point). The values are the number of examples seen (N), the sum of their target values ($\sum y$), as well as the sum of their squared target values ($\sum y^2$) for all examples where the value of the corresponding attribute is equal or smaller than this node's value and again for all examples greater than this node's value. When the leaf is tested for splitting, the attribute value combination with the highest SDR is retrieved. It is necessary to base the splitting decision on a sufficient amount of examples, which is especially important in the incremental setting, where memory consumption should be as low as possible and where the necessary examples are only accumulated over time. To find this valid amount of examples, the FIMT algorithms follows the idea of the VFDT approach and uses a statistical bound. Instead of the Hoeffding bound, which is only useful for the classification task, FIMT uses the Chernoff bound. It is independent of the distribution and gives a relative or absolute approximation of the deviation of a variable X from its expectation μ. It states that the true mean of a random variable X is at least $\bar{\mu} - \epsilon$ with probability $1 - \delta$, where

$$\epsilon = \sqrt{\frac{3\bar{\mu}}{n} ln(\frac{2}{\delta})} \tag{3.16}$$

and n is the number of examples seen so far. In the setting of FIMT, the Chernoff bound states if enough examples have been seen to guarantee with a probability of $1 - \delta$ that the splitting decision (SD_1) with the highest SDR is significantly better than the one (SD_2) with the second highest SDR. Consequently, the variable X, which is evaluated by the Chernoff bound, is the difference between the highest and second-highest SDR

($\Delta SDR = SDR(SD_1) - SDR(SD_2)$). It follows that if $\Delta SDR > \epsilon$, SD_1 is with probability $1 - \delta$ the correct choice. The observed difference is used as the expected average $\bar{\mu}$ in Formula 3.16, as it can be seen as an average over the examples. In the case of very similar values for the best and second best SDR, the Chernoff bound will need too many examples for a decision with high confidence. Ties are broken by the parameter τ, which represents the level of error to be accepted. The current best splitting decision SD_1 is chosen. To further improve the processing speed, each leaf is only tested for splitting every N_{min} examples arriving in this leaf. This is done at the cost of possibly delayed splits. FIMT has no pruning, so that no wrong decisions are corrected and the tree constantly increases in size with every example. Nevertheless, a memory limit parameter can be used to restrict and finally stop tree growth. This approach is implemented in the VFML library and deactivates the least promising leaf nodes under memory shortage. Reactivation is still possible if the memory resources increase again. The models in the leaves are perceptrons with the learning rate η. Concept drift is not detected by FIMT and consequently this algorithm shows its strengths on stationary data distributions.

FIRT FIRT (Fast and Incremental Regression Tree) [69] is nearly identical to the FIMT algorithm. The only difference lies in the used models. FIRT uses the mean target value of all examples in the leaf node as prediction. Consequently, FIRT can be considered as the online version of CART.

FIRT-DD FIRT-DD (Fast and Incremental Regression Tree with Drift Detection) [72] is an advanced version of the FIRT algorithm, which can adjust to concept drift in the data. To do so, each node of the tree holds a change detection unit. If a change is detected, it is assumed that a concept drift happened in the subspace the node belongs to. By that, local concept drift can be detected, and only this region of the tree can be adjusted without recalculating the whole tree. Local concept drifts can even be detected in parallel. The nearer the concept drift is detected to the root, the more global the effect becomes. The concept drift indicator used in the detection units is the absolute loss. The absolute loss is the absolute difference between the prediction (here: $\bar{y}$) when the leaf was initialized and the prediction which is now available in the node. On concept drift, the loss will increase. Consequently, the evolution of the loss is monitored and continuously tested by a Page-Hinkley test for an change pointing to a concept drift. If the difference between the absolute loss and its average is increasing over a predefined limit, the detection unit at the node raises an alarm and an adaptation process is started. This process is similar to the idea used in the CVFDT system. A concept drift in a subspace has the consequence of an outdated subtree. Consequently, it should be pruned. As pruning of a subtree results in an abrupt accuracy decrease, an alternative, more accuracy friendly approach is chosen. When the concept drift is detected in a node, a new tree is learned in parallel on the examples reaching this node. The subtree in the original tree will be replaced by the alternate subtree as soon as it becomes more accurate. Slow progress or decreasing performance of the subtree is a sign of a false alarm or that the subtree cannot achieve better results and the alternate subtree will be deleted. FIRT-DD has no pruning approach, but an adaptation process for concept drift. That means that the algorithm will behave exactly the same way as FIRT on stationary datasets.

FIMT-DD FIMT-DD (Fast and Incremental Model Tree with Drift Detection) [70] is a model tree with extensions to detect and react on concept drift. It is based on FIMT and has the same concept drift adaptation method as FIRT-DD. In comparison to the publication of FIMT, the Hoeffding bound is used instead of the Chernoff bound to validate the best splitting decision. This time, the SDR fraction $SDR(SD_2)/SDR(SD_1)$ with a range $R \in [0,1]$ is calculated. The true average of the variable r ($\bar{r}_{true}$) is now within the ϵ interval: $\bar{r} - \epsilon \leq \bar{r}_{true} < \bar{r} + \epsilon$, where $\bar{r} = \frac{1}{n}\sum_{i=1}^{n} r_i$ and ϵ is the value of the Hoeffding bound $\epsilon = \sqrt{\frac{R^2 ln(1/\delta)}{2n}}$. Again, if after N_{min} examples $\bar{r} + \epsilon < 1$ and, consequently $\bar{r}_{true} < 1$ holds, SD_1 is the best splitting decision.

ORTO ORTO (On-line Regression Trees with Options) [73] is an incremental option tree. Instead of regular split nodes, options are used as splitting decisions in the internal nodes. These options are concatenations of several splitting decisions through OR-functions. ORTO was developed to eliminate the delayed tree extension process produced in Hoeffding-based trees through equally good split decisions. The tie-breaking threshold is not considered as useful as knowledge of the problem domain is needed to set it properly. ORTO is based on the FIRT algorithm. To find the best splitting decision, a new approach is used to introduce option nodes as well: if, after observing N_{min} examples, $\bar{r} - \epsilon$ holds, a normal split is made as already known from the previous algorithms. Otherwise, an option node is introduced including all splitting decisions SD_i for which the inequality $SDR(SD_i)/SDR(SD_1) > 1 - \epsilon$ holds. To restrict excessive tree growth, the number of options are reduced with the depth of the tree. This is considered as a valid approach, as options are thought to be most useful near the root. Predictions are produced by either choosing the tree of the option node with the highest accuracy so far, or by following all trees and choosing the average prediction. As ORTO is based on FIRT, no pruning method is considered.

FIOT FIOT (Fast and Incremental Option Trees) [71] is very similar to ORTO. The main focus of this work is to apply this algorithm to learning problems under gradual concept drift. The current concept is gradually replaced with a new one and the incremental option tree FIOT is developed to handle this transition state. FIOT is similar to ORTO and extended by the concept drift adaptation framework of FIRT-DD. To limit extensive tree growth, only the k best performing trees in the options are stored. If new options arise and more than k trees are available, the worst ones are removed. Predictions are made by either taking the best tree in an option or by calculating the average prediction over all predictions in the option. Again, under no concept drift, no pruning will occur.

Chapter 4

Data Sources

This chapter describes the source of five different data. Three of them have an artificial and two a real-world source. Each data source is used in the following chapters to produce datasets or data streams. The creation process is explained in the chapters itself.

4.1 Artificial Sources

In the following, three artificial data sources are described, which are used in the next chapters to create data streams or datasets.

4.1.1 2DimTree

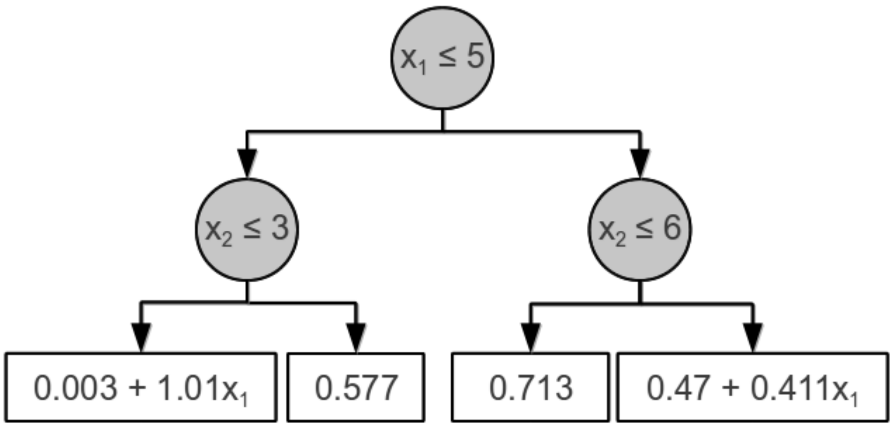


Figure 4.1: 2DimTree dataset generation tree

Examples from a piecewise linear function with only two numerical input variables were generated by the tree introduced by C. Vens *et al.* [135] (cf.

Figure 4.1). An example is created by randomly choosing with uniform probability each input dimension (x_1 and x_2) in the range of [0,10]. Based on the input dimensions, the tree is traversed from the root to the leaf and the model is applied to determine the target variable y.

4.1.2 2DimFunction

Examples from a non-linear function were derived using the same function as already used by Schaal and Atkeson [118] (cf. Equation 4.1).

$$y = \max(e^{-10x_1^2}, e^{-50x_2^2}, 1.25e^{-5(x_1^2+x_2^2)}) + \epsilon \tag{4.1}$$

The input dimensions (x_1 and x_2) are in the range [-1, 1] and randomly chosen with uniform probability. The target variable y is calculated by inserting the input values in equation 4.1. The produced function is shown in Figure 4.2.

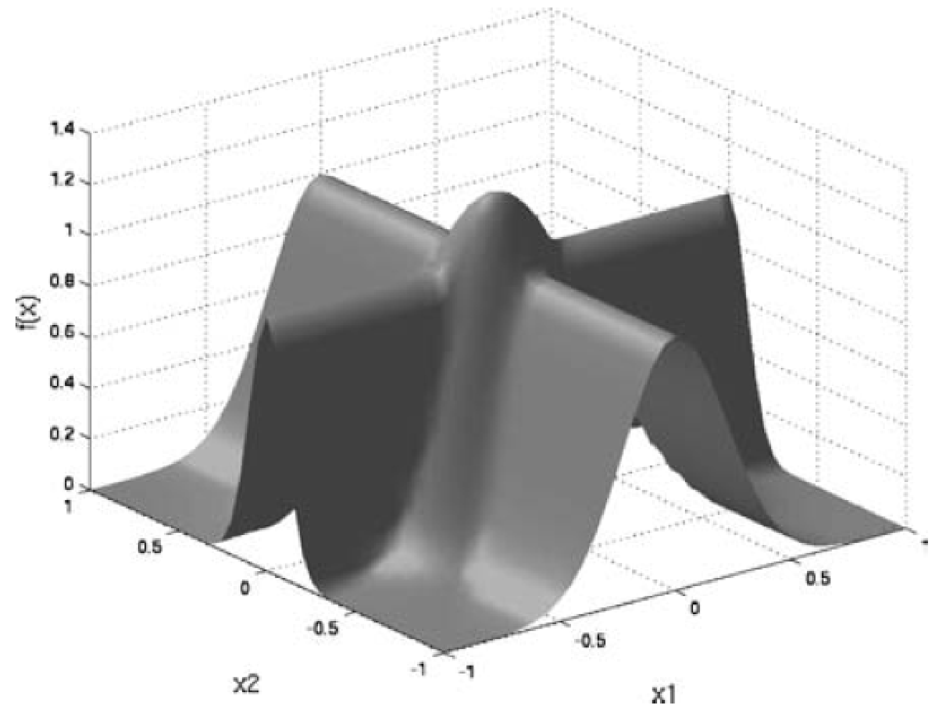


Figure 4.2: Target distribution of 2DimFuntion (Source: [106])

4.1.3 4DimTree

To represent a more complex piecewise linear function, the tree shown in Figure 4.3 was created. All four numerical input dimensions (x_1, x_2, x_3 and x_4) are in the range [0,10] and randomly chosen with uniform probability. Again, the target variable is calculated by traversing the tree from root to the leaf and applying the model to the input values.

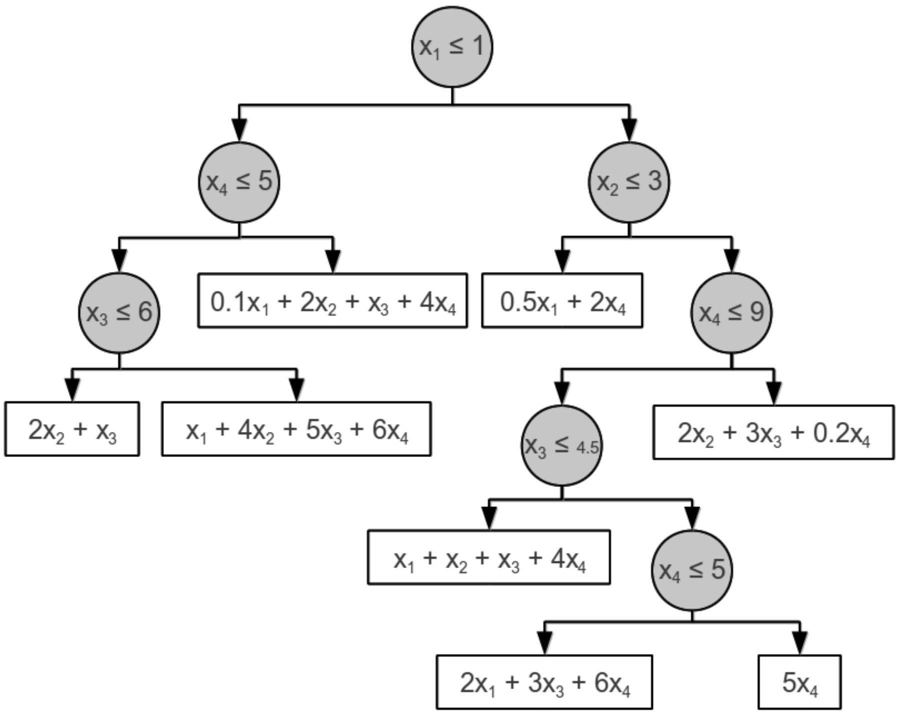


Figure 4.3: 4DimTree dataset generation tree

4.2 Real-World Sources

In the following, two real-world data sources are described, which are used in the next chapters to create data streams or datasets.

4.2.1 Census

The first real-world data source is the US Census Data (1990) dataset from the UCI Machine Learning Repository[1]. It contains 2,458,285 examples and 68 categorical attributes. The dHour89 attribute, indicating the discretized usual hours worked per week in 1989, was chosen as the target variable y.

[1]http://archive.ics.uci.edu/ml/datasets/US+Census+Data+%281990%29

4.2.2 Airline

The second real-world data source is a dataset from the DataExpo09[2] competition. The original data stores 120 million records describing flight arrival and departure details for all commercial flights within the USA, from October 1987 to April 2008. The dataset was preprocessed and erroneous entries were removed. Moreover, entries displaying flights which were canceled or diverted were removed. 35 input variables (6 numeric, 29 binary) were used to predict the minutes the flight can compensate from a departure delay (departure delay - arrival delay). As input variables the DepDelay (departure delay), Distance (distance of the flight), carrier1...10 (if the flight was done by one of the 10 biggest carriers/ 10 binary variables), hourTravelFreq (the relative flight frequency/ traffic in the departure hour), originDepFreq (the relative frequency of flights departing from the departure airport/ the size of the departure airport), destArrFreq (the relative frequency of flights arriving at the destination airport/ the size of the destination airport), planeage (age of the plane), month1...12 (the month of the year the flight is in, 12 binary variables) and day1...7 (the day of the week the flight is in, 7 binary variables) were used. To facilitate the prediction tasks, only flights with a departure and arrival delay within plus / minus an hour (between -60 and 60 minutes) were chosen. It is assumed that the delay is still of concern for these flights. The final datasets contains 40,682,046 examples and the target variable y ranges from -98 to 91 minutes and is normally distributed.

[2]http://stat-computing.org/dataexpo/2009/

Chapter 5

ILMTs - An Experimental Evaluation

5.1 Introduction

Dealing with massive datasets is one of the great challenges faced today, as data has become ubiquitous in all areas of life. Data is either gained over time from data streams, or at one point in time from huge data collecting and generating projects. Examples are, e.g., next generation sequencing (NGS) projects, which double their sequencing capacity of base pairs (bp) per dollar every fifth month [125], the NASA Earth Observation System (EOS), which produces 2.9 TB data per day [24], data concerning the steadily increasing world population[1], or user-generated data from mobile devices, to name just a few prominent examples.

Working with massive amounts of data is in general both time and memory consuming. Traditional batch algorithms had to load the whole dataset into main memory for processing. Alternatively, approaches like sampling [84, 77] or load shedding [26] have been developed to perform the mining task on a suitable subsample of the available data. With the development of incremental or online algorithms to handle the flood of data from a data stream, another valid alternative has become available. Incremental online algorithms are fast and need only a fraction of the memory used by batch algorithms. Therefore, the use of incremental online algorithms for massive data sources is one option that is currently investigated extensively.

One of the most important and best-studied class of predictive data mining algorithms is based on trees. For regression, model trees with linear models in the leafs are often the method of choice as compared to simple regression trees. In the incremental online setting, only few approaches of such incremental linear model trees (ILMT) exist, differing extremely in their degree of complexity. However, surprisingly little is known about

[1]http://data.worldbank.org/

their relative performance, strengths and weaknesses, first, because they have not been evaluated under the same conditions and, second, because only small datasets were considered for their comparison. Thus, it is an open question how different features of the algorithms (as described below) affect the performance on massive real-world datasets, and whether more complex functionality really turns out to be beneficial. The contribution of this chapter is to address this question and shed light on the behavior of features of algorithms from this important class (incremental models trees for massive datasets).

This chapter first gives a compact definition of ILMTs due to their importance. Then, it motivates the usage of online algorithms by a sampling example. Next, it shows the performance of ILMTs on massive stationary datasets by analyzing and comparing the three algorithms *FIMT*, *IMTI-RA* and *IMTI-RD*. One important aspect of this work is that all algorithms are implemented in the same framework to make the results comparable and reduce biases arising from implementation choices. The chapter closes with a discussion.

5.2 Related Work

Currently, only few ILMT algorithms have been developed and the amount of different approaches further decreases if they were applied on stationary data. Add-on features to handle concept drift are not needed in this setting and consequently, these algorithms then have the same functionality as the algorithms they were extended from (e.g. FIMT-DD and FIMT). A detailed description of available ILMTs can be found in Section 3.3.2. This chapter focuses on the comparison of FIMT, IMTI-RA and IMTI-RD as all three algorithms employ splitting criteria of different complexity. To the best of our knowledge, the only comparison of all three algorithms was made in the *FIMT-DD* publication. Unfortunately, the algorithms were implemented in different languages, by different persons, in different frameworks. Furthermore, the algorithms were compared using only one parameter setting on maximally 90,000 examples. For this reason, the relative strengths and weaknesses of various algorithms from this important class of algorithms are currently not known and will be investigated in this chapter, on massive stationary datasets, using different parameter settings, within the same environment.

5.3 ILMT

Linear model trees (LMTs) are well-studied algorithms, applied even to complex non-linear regression problems. Their success lies in the idea to divide the problem into several easier subproblems. We assume a dataset

$D = \{e_1, \ldots, e_n\}$, with examples given as feature vector $\langle x_{i1}, \ldots, x_{ij}, y_i \rangle$. The task is to predict the underlying regression function $\vec{y} = f(\vec{x})$, which can be approximated by a linear model $\hat{y_i} = \sum_{k=0}^{j} \beta_k x_{ik}$. Linear model trees now solve this task by splitting the input space over the input variables $X_1, \ldots, X_j$ into several subspaces, learning a linear model in each one of them. They work on the assumption that arbitrary functions can be approximated by piecewise linear models. More subspaces allow for more accurate approximations. An LMT is a tree consisting of internal nodes *in* and leaf nodes *ln*. Each internal node *in* contains a splitting decision $in.sd_j$ and normally two child nodes *in.left* and *in.right*, which are either leaf or internal nodes. The splitting decision sd_j is based on an attribute X_j and one or several thresholds *th* of the attribute value on which the input space is partitioned. The specification of the splitting decision can be different, depending on the type of the chosen attribute. Splitting decisions on categorical attributes have thresholds consisting of subsets of the categorical values. For numerical attributes, the thresholds are either specific numerical values or ranges. Leaf nodes *ln* contain no children, but a linear model (*ln.m*) used for prediction. Examples receive a prediction by traversing the tree from the root to a leaf and applying its model on the example. Standard LMTs are learned on a dataset that is completely available at training time and small enough to be processed in main memory (batch learning). In contrast, ILMTs are updated incrementally, processing only one example e_i after another from a possibly infinite data source, which is advantageous in terms of processing speed as well as memory usage. Each example is used only once to further adjust the ILMT and discarded afterwards due to assumed memory limitations. Consequently, as no examples are stored, no group of examples can be queried to find the best possible splitting decision sd_j for a leaf at the appropriate time of split. That is why aggregated statistics over the examples observed in the leaf are stored in each leaf node. This split statistic (*ln.spst*) stores characteristics of possible splitting possibilities for the leaf node *ln*, from which the best splitting decision sd_j can be extracted. Different ILMT algorithms store different characteristics in *spst* and use different split evaluation methods. These and other algorithm characteristics for ILMT algorithms are explained in detail in Section 3.3.2.

5.4 Experimental Evaluation

In this section, the performance of the three ILMTs on massive stationary datasets is evaluated. First the evaluation setting is explained, followed by a description of the chosen datasets. Then, a motivating example is presented and subsequently, the application results of the online algorithms are shown.

5.4.1 Evaluation Setting

All three ILMT algorithms are reimplemented within the same framework using Java. As many methods as possible are shared by the algorithms. By that, the results are comparable and effects based on the language bias as well as on the programming skills are reduced. FIMT was reimplemented according to the published information [69]. The IMTI-RD and IMTI-RA implementations were modeled after the original implementations provided by the authors. Each evaluation was run as a single thread on a Quad-Core AMD Opteron Processor 8384 with 2.6 GHz. Each Java process was given 2 GB of RAM. To address the dependency of the online algorithms on the examples' ordering, each result value is the average over five runs. To judge the performance of the given algorithms on massive datasets, the runs were evaluated whether they run quickly, have a low MAE (mean absolute error) and a low RAM consumption.

5.4.2 Datasets

The evaluation is done on three different data sources: 2DimTree, Census and Airline. The 2DimTree data source was used to extract five separate training sets of 50,000,000 examples and five test sets with a size of 1,000,000 examples. The airline dataset (source) was used to create five training sets with 39,682,046 examples with five test sets with 1,000,000 examples each. This was done by rearranging the original dataset to avoid intrinsic concept drift. The census source was used to create five test sets by randomly choosing 500,000 examples from the whole set without replacement. The respective remainders of the set were used as the basis for generating each training set by repeated oversampling. Five such census training sets were created, each comprising 50,000,000 examples.

5.4.3 Online - Batch Comparison

To process massive stationary datasets under RAM constraints, a standard method is to select only a subsample of the original dataset to train the model. To show the limitations of batch algorithms on massive datasets, we first show how a batch algorithm using a sampling method performs on the described datasets. This is done by using the Weka-Workbench Version 3.7.5, which is also implemented in Java and thus ensures approximately comparable runtime measurements. For each process, the Java environment received 2 GB of RAM. As the *M5* algorithm is the batch version of the *FIMT* algorithm, the implementation in Weka (*M5P* [140]) is used with default parameters. Smoothing and pruning are deactivated, as they are also not used in *FIMT*. For the sampling method, the reservoir sampling algorithm "R" [136] was chosen. For every dataset, *M5P & R* is trained with each of the five training sets and tested on the corresponding test set.

Table 5.1: Algorithm performance after the maximal amount of processable examples

Dataset	Algo	run time (sec)	MAE	MBytes
2DimTree	M5P & R	463	0.0173	2,048.0
	FIMT	233	0.0114	11.5
airline	M5P & R	2,689	10.9	2,048.0
	FIMT	2,232	8.2	149.8
census	M5P & R	2,756	0.3721	2,048.0
	FIMT	2,824	0.3126	60.7

The results are averaged over the five runs. Using 2 GB of RAM, only a very small subsample of the whole dataset can be used by the batch learner to train the model. Depending on the complexity of the dataset, the amount of examples ranges from 3.3 million (2DimTree), over 0.7 million (airline) to only 0.5 million examples (census). In contrast to the batch learner (*M5P & R*), the online learner (*FIMT*) is able to process all examples from the datasets. Table 5.1 shows the performance of *M5P & R* after the maximal amount of processable examples for each dataset as well as the performance of the online learner *FIMT* ($N_{min} = 700$) after processing all examples from the datasets. It can be seen that the memory consumption of the online learning approach is much smaller than for the batch approach. While the batch algorithm consequently uses 2 GB of RAM, the online learner uses maximally about 150 MB of RAM. Additionally, as the model of the batch algorithm is learned with a much smaller dataset, the MAE of the batch algorithm is consequently worse than the one of the online learner. If we would further restrict the batch approach to the same amount of RAM as needed by the online learner (max. 150 MB), the accuracy gap would even increase. But not only the predictive performance is better, even processing the whole dataset takes less time. The only exception is the census dataset, where its complexity results in a longer calculation time of negligible 68 seconds for the online algorithm. Overall, it can be stated that the online algorithm extracts a better model from the dataset by using only a fraction of the RAM with faster or nearly equal processing times.

5.4.4 ILMT Performance Evaluation

This section shows the performance of *FIMT*, *IMTI-RA* and *IMTI-RD* on the three massive stationary datasets using different parameter settings. At first the chosen parameter setting is explained, followed by a description of the evaluation method. Finally, the results are presented and interpreted.

Table 5.2: Run IDs and their parameter setting

	FIMT			
ID	1	2	3	4
N_{min}	100	300	500	700

	IMTI-RA		IMTI-RD					
ID	5	6	7	8	9	10	11	12
π	0	0.025	0	0	0	0.025	0.025	0.025
κ			3	5	10	3	5	10

Parameter Settings

For the *FIMT* algorithm, all runs were based on the parameter setting considered as useful in the original publication: $\delta = 1\text{x}10^{-6}$, $\tau = 0.001$, $N_{min} = 300$ and $\eta = 0.001$. *IMTI-RA* and *IMTI-RD* are based on the default parameter settings proposed in the original implementation: $\pi = 0.025$, $\alpha_{split} = 0.0001$, $\alpha_{prune} = 0.001$, $\delta_0 = 0$, $\kappa = 5$, and $N_{MinModel} = 3*$number of input dimensions. Due to space limitations, we focus the evaluation on the parameters with the highest influence on the induction complexity and model performance: N_{min}, κ and π. The effect of N_{min} (the split test frequency) for *FIMT* is tested over the values 100, 300, 500, and 700. For *IMTI-RD*, the number of candidates (κ) is evaluated over the values 3, 5, and 10, and smoothing (π) was either disabled ($\pi = 0$) or activated with a value of 0.025. Smoothing was deactivated for all runs on the 2DimTree dataset, as the decision function is clearly piecewise linear and smoothing is not needed. Thus, there are no results for the runs with ID 6, 10, 11, and 12 on the 2DimTree dataset. An overview of the different parameter settings is given in Table 5.2, and the introduced ID is used in all further analyses. For the parameters not listed, the above mentioned default settings are used.

Evaluation Method

To simulate also smaller training sets and to gain insight into the dependency of the algorithms' performance on training set sizes, each run was periodically interrupted and tested on the corresponding test set. The run time needed to test the model is not included in the final run-time results.

Results

Tables 5.3, 5.4 and 5.5 present the results of the different parameter settings on each dataset. All results of *FIMT* and *IMTI-RA* are calculated for all datasets on the whole dataset size. The *IMTI-RD* results are based on 5.0 million training examples for the airline and 2DimTree datasets and on 1.5 million examples for the census dataset. This is due to the low example processing speed of *IMTI-RD*, a consequence of its high intra-algorithmic complexity. That means that in the worst case only 2.1 examples per second

Table 5.3: Final mean (stdev) results on the whole 2DimTree dataset (ID 7-9: after 5 mio. examples)

ID	time in h	MAE	MB	# Leaves
1	0.08 (0.002)	0.0116 (0.0011)	12 (0.4)	794 (14.2)
2	0.07 (0.002)	0.0116 (0.0011)	12 (0.4)	704 (32.7)
3	0.07 (0.002)	0.0109 (0.0006)	12 (0.5)	674 (32.9)
4	0.06 (0.001)	0.0114 (0.0014)	11 (0.5)	679 (42.0)
5	0.59 (0.049)	0.0006 (0.0003)	7 (1.6)	3499 (803.1)
7	0.06 (0.007)	0.0008 (0.0001)	2 (0.2)	338 (47.6)
8	0.08 (0.006)	0.0007 (0.0001)	2 (1.1)	335 (167.6)
9	0.13 (0.011)	0.0007 (0.0001)	3 (1.2)	283 (106.2)

Table 5.4: Final mean (stdev) results on the whole airline dataset (ID 7-12: after 5 mio. examples)

ID	time in h	MAE	MB	# Leaves
1	0.7 (0.01)	8.2405 (0.0084)	311 (1.1)	25497 (170.1)
2	0.7 (0.01)	8.2427 (0.0091)	211 (3.2)	15364 (132.3)
3	0.6 (0.01)	8.2441 (0.0105)	171 (0.7)	11878 (42.1)
4	0.6 (0.01)	8.2450 (0.0094)	150 (1.1)	10061 (57.7)
5	21.1 (1.0)	8.3448 (0.0094)	12 (1.1)	405 (41.8)
6	20.9 (1.0)	8.3416 (0.0095)	12 (1.1)	405 (41.8)
7	25.2 (1.3)	8.3728 (0.0095)	4 (3.1)	2 (1.3)
8	41.7 (0.7)	8.3747 (0.0082)	5 (1.9)	1 (0.5)
9	77.9 (2.4)	8.3747 (0.0085)	12 (9.3)	2 (1.3)
10	25.2 (1.3)	8.3727 (0.0095)	4 (3.1)	2 (1.3)
11	41.8 (0.3)	8.3747 (0.0083)	5 (1.9)	1 (0.5)
12	77.6 (2.4)	8.3748 (0.0085)	12 (9.3)	2 (1.3)

can be processed by *IMTI-RD* on average for the census dataset (cf. Table 5.5 ID 12).

First of all, the results for different parameter values of each algorithm are compared. Focusing on the N_{min} parameter of the *FIMT* algorithm (ID 1-4), it can be observed that lower values result in bigger trees, needing more RAM and processing time. This is logical, as each leaf is tested more frequently for splitting for a smaller value of N_{min}. Taking the MAE into account, it can be seen that the investment of time and memory does not pay back, as the prediction gain is only marginal. High values for N_{min} show comparable MAE results, while at the same time using much less memory and calculation times for all given datasets. This suggests that for massive datasets larger values for N_{min} should be chosen. The same can be observed for the number of candidates used by the *IMTI-RD* algorithm (ID 7-9 and 10-12). For all datasets the resulting MAE is very similar, while the final runtime is up to three times as high for $\kappa = 10$ compared to $\kappa = 3$. Another interesting fact is that *IMTI-RD* produces very small trees for the amount of examples observed. Such a feature may be very interesting when a compact and easy-to-interpret model is desired. Additionally, the smoothing parameter was varied to evaluate its usefulness. Tree size and memory usage is

Table 5.5: Final mean (stdev) results on the whole census dataset (ID 7-12: after 1.5 mio. examples)

ID	time in h	MAE	MB	# Leaves
1	0.9 (0.02)	0.3140 (0.0013)	116 (1.2)	6414 (54.6)
2	0.8 (0.02)	0.3131 (0.0010)	79 (0.6)	3891 (21.3)
3	0.8 (0.01)	0.3128 (0.0011)	67 (1.0)	3139 (63.7)
4	0.8 (0.01)	0.3126 (0.0011)	61 (1.8)	2765 (92.2)
5	161.1 (10.4)	0.3604 (0.0053)	920 (131.4)	10707 (1497)
6	161.1 (9.0)	0.3866 (0.0101)	920 (131.4)	10707 (1497)
7	61.2 (3.8)	0.3421 (0.0102)	66 (23.7)	5 (1.8)
8	103.0 (4.6)	0.3251 (0.0038)	192 (66.5)	10 (3.5)
9	211.2 (8.0)	0.3253 (0.0036)	437 (193.1)	11 (4.9)
10	62.6 (2.1)	0.4061 (0.0636)	66 (23.7)	5 (1.8)
11	104.9 (5.4)	0.6504 (0.1660)	192 (66.5)	10 (3.5)
12	196.1 (28.1)	0.3272 (0.0034)	436 (194.5)	11 (4.9)

not changed by the parameter, as only the prediction method is changed. For that, only the MAE and the run time is influenced by the parameter. Comparing the *IMTI-RA* runs 5 and 6 on the airline dataset shows that smoothing slightly increases the prediction accuracy. On the other hand, it seems more harmful on the census dataset. Nevertheless, choosing the right parameter setting could further improve the model.

After having examined the results for the maximal dataset sizes, it is also interesting to reason about even larger datasets. Therefore, the development of the MAE, the runtime and the RAM usage is plotted (cf. Figures 5.1, 5.2, 5.3) for increasing dataset sizes. The development of the corresponding quality criterion can be investigated and a forecast for even larger datasets can be done. As the analyses up to now indicate that parameter settings leading to a simpler induction process do well, the following comparisons are based on the runs with ID 4, 5, and 7. On the 2DimTree dataset, IMTI-RD and IMTI-RA show nearly the same behavior. They both come to a very good MAE after seeing only a few examples, although IMTI-RD outperforms IMTI-RA. However, including the remaining instances from the dataset improves the final MAE only marginally, while the RAM consumption increases steadily. In contrast, the FIMT algorithm further makes use of additional instances as the MAE decreases continuously, which is due to an increasing model size (the RAM consumption increases). Also, FIMT is much faster than the other two. For the airline dataset, the FIMT algorithm has a lower MAE, runs faster and steadily improves its prediction accuracy while it stays constant for IMTI-RD and IMTI-RA. However, the memory consumption is again the largest for the FIMT. For the census dataset, FIMT turns out to be a clear winner, as it is the fastest, most accurate, and the algorithm with the lowest memory consumption. These numbers indicate that for even larger datasets it would be wise to use the FIMT algorithm, with a large N_{min}, which steadily and quickly improves the prediction accuracy. To sum up, choosing the best algorithm depends on the dataset: If the dataset is clearly piecewise linear, IMTI-RA and IMTI-RD come to good results already after observing only few examples. Moreover, if the dataset is rather small (only some 10,000 instances) then IMTI-RD can be considered as a good choice due to its good MAE results. But if larger datasets are considered, then its run time makes its application infeasible. Surely, for more complex piecewise linear datasets, the needed number of examples will increase. Nevertheless, real-world datasets are normally very complex and not necessarily piecewise linear. For this kind of datasets, the FIMT algorithm is by far the fastest solution, building the best model which also still improves with larger datasets. However, the RAM consumption may remain one possible drawback, when the numerical attributes in the datasets hold too many distinct values and thus the E-BST in the split statistic becomes prohibitively large (see the airline dataset). All algorithms in common is that increasing the algorithm complexity is more harmful than useful. Consequently, choosing the best algorithm and parameter for real-world datasets should, according to the previous experiments, follow the rule: Keep it simple, keep it fast [64]!

5.5 Conclusion

This chapter systematically studied, for the first time, the performance of an important class of algorithms, incremental linear model trees, on massive stationary datasets in three different dimensions: prediction error, running time, and memory consumption. The algorithms were tested within the same framework on large-scale artificial and real-world datasets, under various parameter settings. The results indicate that, first, using parameter settings that lead to simpler induction processes result in equivalent MAE in the long run and also come with the advantage of smaller running times. Additionally, on real-world datasets the algorithm with the simplest induction process, FIMT, is the fastest and most accurate algorithm. Its advantage is also still increasing with bigger datasets. Therefore, our experiments suggest that simplicity is a virtue when learning incremental linear model trees on massive datasets. As a consequence, an extension of the FIMT algorithm would be interesting with a careful limitation of memory usage. This could be either done by optimizing the split statistic storage or by developing a pruning technique that further keeps the tree small and fast. Finally, a similar study on drifting data streams, using FIMT-DD instead of FIMT, should reveal further interesting insights.

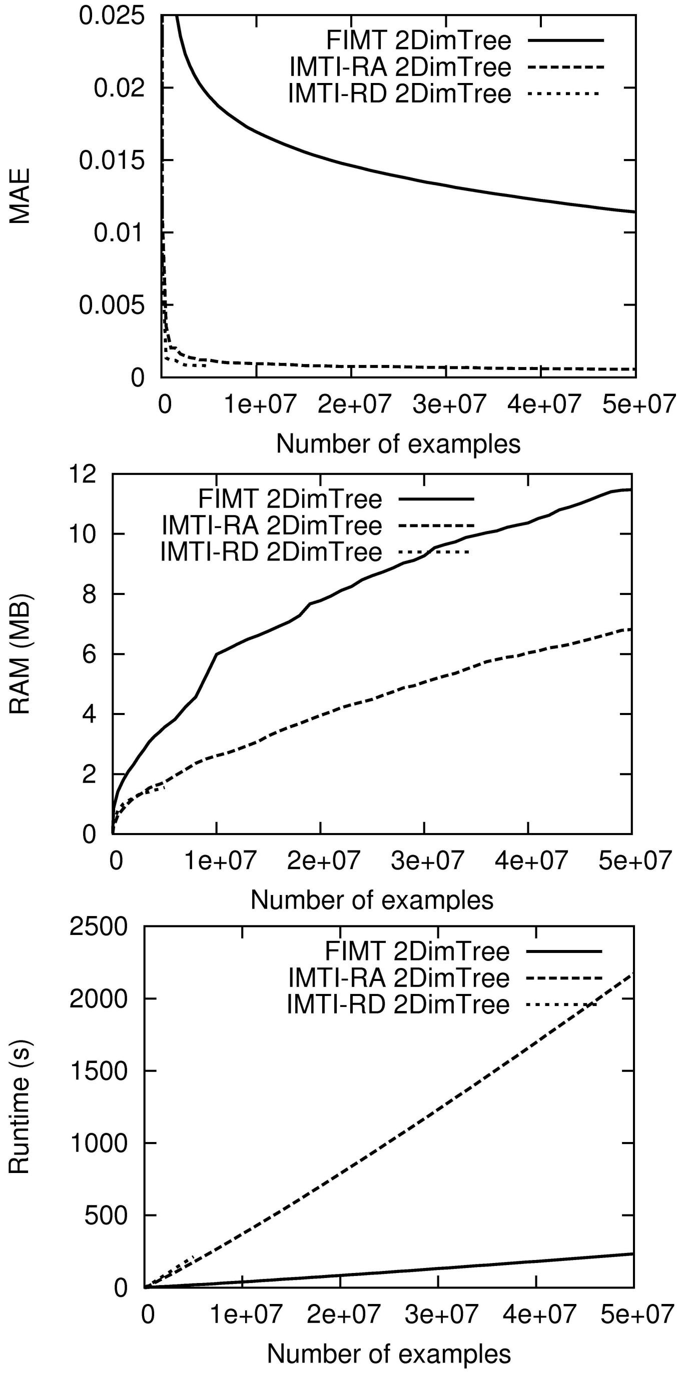


Figure 5.1: Development of the MAE, runtime and memory consumption for the 2DimTree dataset and each algorithm using the parameter setting leading to the least complex induction process.

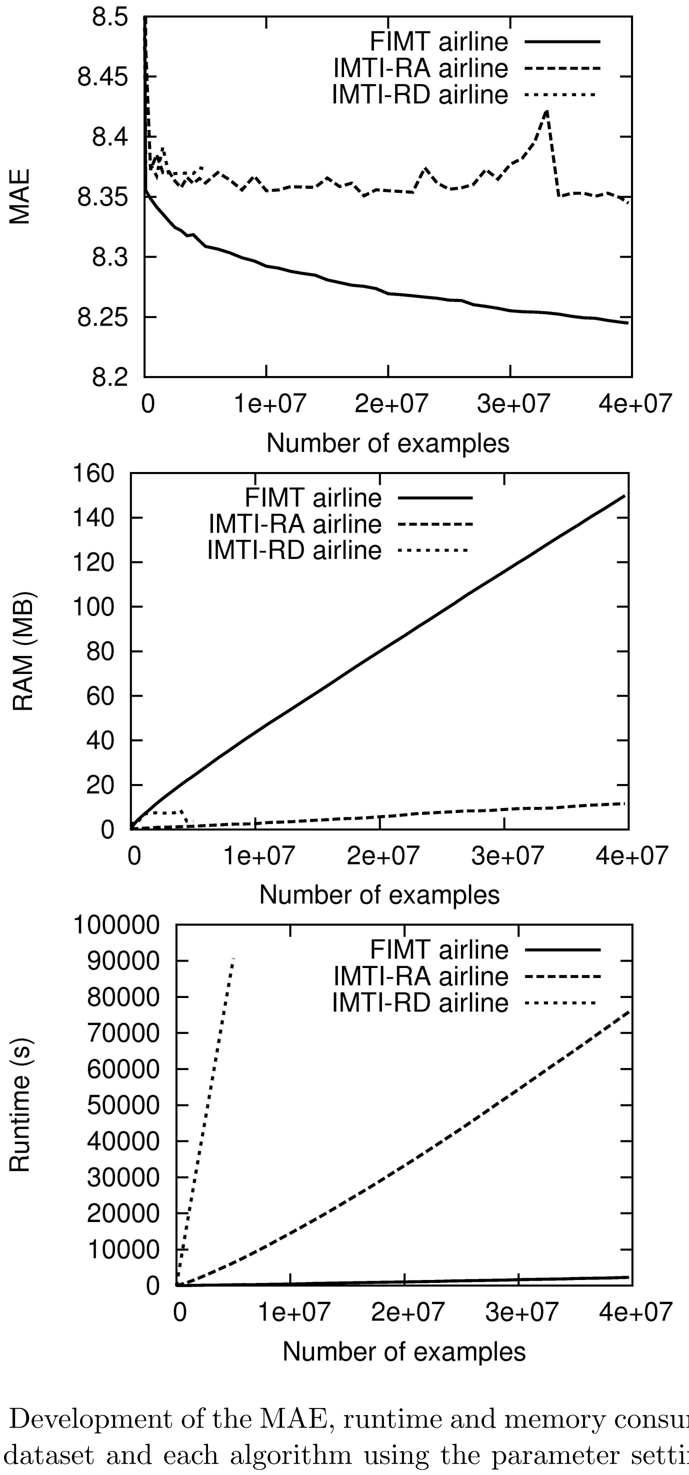


Figure 5.2: Development of the MAE, runtime and memory consumption for the airline dataset and each algorithm using the parameter setting leading to the least complex induction process.

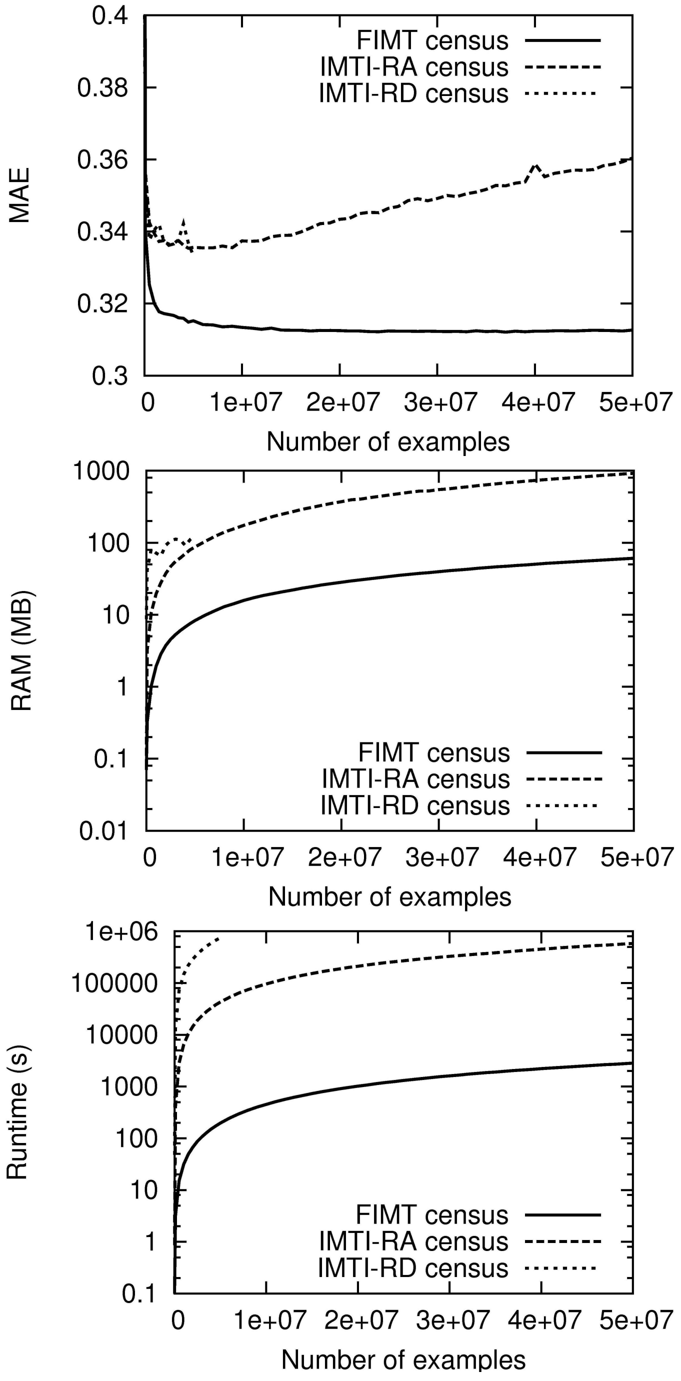


Figure 5.3: Development of the MAE, runtime and memory consumption for the census dataset and each algorithm using the parameter setting leading to the least complex induction process.

Chapter 6

Pruning ILMTs with Approximate Lookahead

Many applications generate abundant data: Biological high-throughput experiments, sensor applications, network monitoring and traffic management, process and quality control in manufacturing, email, news, and blogging systems, to mention just a few examples. Analyzing data from such data sources poses several challenges for data mining, as the produced datasets can either be too large to be processed in main memory at once or the data can constantly grow over time (data streams). As a consequence the data stream model has been developed, where all algorithms must process data incrementally, given strictly limited resources. Instances are processed one at a time, using a limited amount of memory and time.

This chapter focuses on pruning incremental linear model tree (ILMT) algorithms. Linear model tree algorithms split the instance space recursively, basing their split decisions on estimated prediction gains resulting from linear models in the subspaces. Instead of expensive precise lookahead, linear model tree algorithms usually employ some form of approximate lookahead. Approximate lookahead saves time and space, but may choose sub-optimal splits, which in turn may lead to overly large trees. Unnecessarily large trees both waste space and time, as they generally slow down instance processing. Pruning techniques promise to remedy this problem by detecting and removing subtrees that do not provide significant improvements. This chapter develops and evaluates a pruning technique for incremental linear model trees with approximate lookahead on stationary data sources. As massive datasets can be seen as limited stationary data streams in the framework of incremental learning, both are used equivalently in the following. Tests on five large stationary datasets shed some light on the trade-off between tree complexity/processing speed and prediction error. Furthermore, the performance of the incremental online learning approach is compared with an equivalent batch approach to highlight the applicability to massive datasets.

The chapter is organized as follows: First, the concept of approximate lookahead is explained in combination with incremental linear model trees. Then related work is surveyed. In the next Section, the pruning algorithm GuIP is described, which is evaluated on five large datasets in the following. After that, its applicability to stationary data streams and massive datasets is highlighted, and finally a conclusion is given.

6.1 ILMT with Approximate Lookahead

Let a dataset $D = \{e_1, \ldots, e_n\}$ be given, where each example e_i is represented as a vector $< x_{i1}, ..., x_{ij}, y_i >$. The task is to predict the underlying regression function $\vec{y} = f(\vec{x})$, which can be approximated by a linear model $\hat{y_i} = \sum_{k=0}^{j} \beta_k x_{ik}$. Linear model trees (LMT) solve this task by splitting the input space over the input variables $X_1, ..., X_j$ into several subspaces, learning a linear model in each one of them. They work on the assumption that arbitrary functions can be approximated by piecewise linear models. More subspaces allow for more accurate approximations. An LMT is a decision tree consisting of two types of nodes n, either internal nodes in or leaf nodes ln. Each internal node in contains a splitting decision $in.sd_j$ and normally two child nodes $in.left$ and $in.right$, which are either leaf or internal nodes. The splitting decision sd_j is based on an attribute X_j and one or several thresholds th of the attribute value on which the input space is partitioned. The specification of the splitting decision can be different, depending on the type of the chosen attribute. Splitting decisions on categorical attributes have thresholds consisting of subsets of the categorical values. For numerical attributes, the thresholds are either specific numerical values or ranges. Leaf nodes ln contain no children, but a linear model ($ln.m$) which is able to be updated incrementally (e.g., by the *perceptron* [116] or RLS algorithm) and is used for prediction. Standard LMTs are learned on a dataset that is completely available at training time and small enough to be processed in main memory (batch learning). In contrast, ILMTs are updated incrementally, processing only one example e_i after another from a possibly infinite data source DS ($e_i \in DS$), which is advantageous in processing speed as well as in memory usage. Each example is used only once to further adjust the ILMT and is discarded afterwards due to assumed memory limitations. Consequently, as no examples are stored, no group of examples can be queried to find the best possible splitting decision sd_j for a leaf at the appropriate time of split. That is why aggregated statistics over the examples observed in the leaf are stored in each leaf node. This split statistic ($ln.spst$) stores characteristics over all possible splitting possibilities for the leaf node ln from which the best splitting decision sd_j can be identified. Different ILMT algorithms store different characteristics in *spst* and use different split evaluation methods. Optimally, with each possible split in the statistic, the actual linear models, which would be created using this split as the best splitting decision sd_j, are maintained and evaluated. Basing the decision for the best split on these actual linear models in the respective subspaces is referred to as *precise lookahead* in this work. Unfortunately, using the precise lookahead approach over all attributes X_j for every observed threshold th at each leaf is expensive with regard to both time and memory. Thus simplifications are needed. One simplification is to limit the number of possible thresholds th for each attribute X_j in $spst$, but to still maintain all the actual linear subspace models. More common simplifications still consider all possible splits, but do not rely on the actual linear subspace models. Usually, a more efficient approximation method is used. The most efficient one bases its splitting decision on the maximal reduction of the standard deviation of the target variable y. Algorithms using any efficient approximation method as lookahead are referred to in this work as ILMTs with *approximate lookahead*. Depending on the type of lookahead, the models in the newly created leaves are initialized differently when a split is made. As mentioned above, in the precise lookahead setting, the actual submodels for each possible split are stored in $ln.spst$. When the leaf ln is now split using the possible split as the splitting decision sd_j, the corresponding actual models can be used as starting models for the newly created child leaves. In the approximate lookahead setting, no actual models can be passed as none are learned in $ln.spst$. Anyhow, to give the new leaves the best possible start, the former leaf model $ln.m$ itself can be passed to the new child leaves as a first approximation.

6.2 Related Work

While many incremental tree induction algorithms exist for the classification task (see Section 3.3.1), only few algorithms are available for the regression task [4]: FIRT, FIMT, FIRT-DD, IMTI-RD, IMTI-RA, IMTI-RD, ORTO and FIOT (see Section 3.3.2 for detailed information). To the best of our knowledge, no pruning technique focusing on the prediction error has been developed so far for ILMTs with approximate lookahead on stationary data sources. IMTI-RA uses a pruning technique based on its splitting decision. Instead of using the actual error reduction in the subtree, pruning is based on a statistical test over the distributions of the positive and negative residuals. FIRT and FIMT do not use pruning. Its extensions FIRT-DD and FIMT-DD replace or remove outdated subtrees when concept drift is detected. Nevertheless, on stationary data sources, where concept drift is absent, the pruning method used by FIMT-DD is not triggered, and therefore the generated trees are the ones generated by the unmodified FIMT algorithm. ORTO and FIOT are adaptations of the afore mentioned algorithms to form option trees and also have no pruning component.

6.3 Guarded Incremental Pruning / GuIP

GuIP [63] is an incremental pruning method specialized on ILMTs with approximate lookahead on stationary data sources. It prunes the tree after each training example and therefore keeps the tree in a pruned state at all times. To explain this pruning approach, the adaptations for ILMTs with approximate lookahead are shown first. Then the concept of what we call the *prune guard* and the pruning decision are explained.

6.3.1 Adaptations

GuIP can be used on any ILMT with approximate lookahead. To do so, the ILMT specific properties mentioned in Section 6.1 have to be extended. First of all, linear models are needed in each node (leaf nodes $ln.m$ as well as internal nodes $in.m$), as their performance is compared for pruning. To measure performance, model statistics need to be maintained together with each linear model.

Table 6.1: Node properties: Bold entries are extensions to each ILMT needed by GuIP

in	internal node
$in.sd_j$	splitting decision
$in.left$	left child node
$in.right$	right child node
$\boldsymbol{in.m}$	linear model
$\boldsymbol{in.m.RSS}$	prequential error of $in.m$ in the subspace
$\boldsymbol{in.m.N}$	number of examples observed by $in.m$ in the subspace
$\boldsymbol{in.RspIL}$	resplit ignore list

ln	leaf node
$ln.spst$	split statistic
$ln.m$	linear model
$\boldsymbol{ln.m.RSS}$	prequential error of $ln.m$ in the subspace
$\boldsymbol{ln.m.N}$	number of examples observed by $ln.m$ in the subspace
$\boldsymbol{ln.RspIL}$	resplit ignore list

The model statistics consist of two values accumulated over the observed examples in the subspace: The sequentially accumulated model prediction error (prequential error [31], explained later in detail / interleaved test-then-train approach [14], $m.RSS$) as well as the number of examples observed by the model ($m.N$). Once a split decision has been found wrong and consequently been pruned, it is possible that later the very same split decision will again be considered. As we assume stationary data sources, we avoid

Algorithm 5 CalculateDivergence(Internal node *in*, leaf node *ln*)

```
1: $iDivergence := ln.m.N$;
2: $in_{current} := ln.parent$;
3: while $in_{current} \neq in$ do
4:     $iDivergence := iDivergence + in_{current}.m.N - in_{current}.left.m.N - in_{current}.right.m.N$
5:     $in_{current} := in_{current}.parent$
6: end while
7: return $iDivergence$
```

Algorithm 6 ActivePruneGuard(Internal node *in*, Int ξ)

```
1: for all $ln_j \in in.subLeaves$ do
2:     if CalculateDivergence($in$, $ln_j$) $< \xi$ then
3:         return true
4:     end if
5: end for
6: return false
```

"resplits" using the same split decision by storing all splits made at a node in its resplit ignore list *RspIL*. Table 6.1 summarizes all node properties. Bold entries highlight the extensions needed for the pruning approach.

6.3.2 Prune Guard

Pruning ILMTs has to be done with care, as the subleaves' models have to be given the opportunity to adapt to the subspace and to further improve their predictions before being pruned. Using ILMTs with precise lookahead, this requirement is automatically fulfilled, as all subleaves' models have already had enough time to adapt to the subspace when being learned in the split statistic. When chosen as the models in the newly created leaves, the models are well adapted and pruning can be performed without any concern. That is not the case for ILMTs with approximate lookahead. As the models are inherited from the parent node during the splitting procedure, it is crucial to allow the new models to adapt to their subspaces over time. The intuitive approach to wait for all subleaves' models until they have observed more than ξ examples in its specific subspace before pruning is not possible in the incremental setting, because the tree is constantly split. As each split results in a new adaptation process in the new leaves, one cannot ensure that this condition is ever met. For that reason, the adaptation progress is calculated for the subleaves' models since their separation from the subtree root node *in*. This progress is the divergence of the leaf model from node *in* and is measured by the number of examples observed by the model *ln.m* after separation from node *in*. This reflects the number of examples observed by

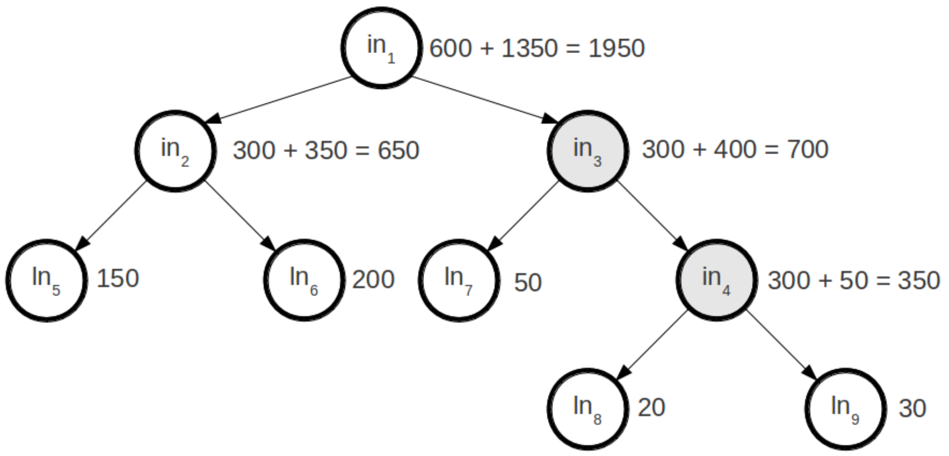


Figure 6.1: Example tree for the prune guard usage. The first summand shows $m.N$ at the moment of the split, and the second summand shows the examples seen in this node after the split. Summing over both equals $m.N$ at the time of the snapshot. Internal nodes with activated prune guard are highlighted in gray.

the model and used to further adapt to the new subspaces since separation.

In the pruning step, after every single training example, any internal node in is prevented by the so-called *prune guard* from being pruned if a model in any subleaf $ln.m$ of its subtree has not yet diverged far enough from that internal node. The pseudocode for the calculation of the divergence of the model in leaf node ln from the internal node in is shown in Algorithm 5. If this number is less than ξ for any subleaf, the prune guard is activated and the node must not be pruned (ref. Algorithm 6).

Example: Figure 6.1 shows an example for prune guard usage. At the time of the snapshot, the tree was learned with 1950 examples (count in the root node). Now assume each subleaf model must have diverged from the comparison node by at least 150 examples ($\xi = 150$). Furthermore, assume node in_4 is being considered for pruning. Calculating the divergence of leaves ln_8 and ln_9 from in_4 shows that the models have only diverged by 20 and by 30 examples. So the prune guard is activated and prevents node in_4 from being pruned and gives the leaves' models further opportunity to improve. The pruning decision of node in_3 is a little bit more complicated, as not all subleaves are direct children. Model $in_3.m$ was adapted to the subspace with 300 examples before the node was split. After that $in_3.m$ has observed 400 examples, of which 50 examples were passed down to leaf ln_7 and 300 examples to node in_4. As in_4 was also split after 300 examples, the remaining 50 examples were passed to the leaves ln_8 (20) and ln_9 (30). As the models are passed to the children after each split, $ln_8.m$ has observed 320 examples since the divergence from node in_3. 20 examples since the model has diverged from in_4, and 300 more since that model has diverged from in_3. Calculating the divergence for all the other subleaves shows that $ln_9.m$ diverged by 330 examples and $ln_7.m$ by only 50 examples. As a divergence of only one subleaf being smaller than ξ is sufficient, the prune guard also blocks in_3 from being pruned. Examples for nodes that can be tested for pruning are in_2 and in_1, because all submodels have diverged far enough and the prune guard is therefore not activated.

6.3.3 Prune Decision

Once the prune guard allows for an internal node in to be tested for pruning (i.e. the prune guard is not activated), a decision is needed whether pruning is advantageous for the tree prediction. This is done by comparing the model performance in node in with the performance of the models in all its subleaves ln_j. Performance is judged by the capability of a model to adapt to its subspace, which is reflected by the model statistics ($m.N$ and $m.RSS$). $m.RSS$, the prequential model error, is measured by the sequentially updated residual sum of squares (RSS), which represents the accumulated squared prediction error:

$$RSS_i = RSS_{i-1} + (y_i - \hat{y_i})^2 \tag{6.1}$$

In the remaining, the following simplifications are used: $RSS_{in} = in.m.RSS$, $RSS_{ln_j} = \sum_{j=1}^{q} ln_j.m.RSS$, $N_{in} = in.m.N$, $N_{ln_j} = \sum_{j=1}^{q} ln_j.m.N$, where ln_j are the q subleaves of node in. Testing node in for pruning, it is evaluated whether the subtree rooted at in harms prediction performance. This is the case if the q subleaves' models $ln_j.m$ have a higher combined prequential error than $in.m$. This means that a negative error reduction (ErrRed, ref. Equation 6.2) using the subtree rooted with in is present.

$$ErrRed(in) = RSS_{in} - RSS_{ln_j} \tag{6.2}$$

In this case, if $ErrRed(in) \leq 0$, pruning would be beneficial. As in the incremental setting the leaves' models have seen fewer examples in their subspaces than the models in the internal nodes, RSS_{ln_j} is assumed to be smaller than RSS_{in}. Consequently, ErrRed(in) will be mostly positive, which, in the incremental setting, does not necessarily mean that the subtree is beneficial for the overall tree prediction. For these cases, an additional evaluation measure is needed which incorporates prediction errors based on different numbers of examples. The Chow test [27, 44] in its extended version [106] , which is used to test if the RSS over the q subleaves ln_j is significantly smaller than the RSS in node in, allows for such different subset sizes:

$$F_{Prune} = \frac{(RSS_{in} - RSS_{ln_j})(N_{ln_j} - qd)}{RSS_{ln_j}(N_{in} - N_{ln_j} + (q-1)d)} \tag{6.3}$$

In this equation, d represents the input dimensionality of each linear model. The null hypothesis states that the underlying regression function is linear in node in, i.e., no subtree is justified. If the alternative hypothesis is true, the RSS reduction is significant, and splitting and growing a subtree is justified. Under the null hypothesis, F_{Prune} is distributed according to the F-distribution with $N_{in} - N_{ln_j} + (q-1)d$ and $N_{ln_j} - qd$ degrees of freedom. An associated p-value greater than a given threshold α_{prune} stands for a non-significant RSS reduction, and consequently the node should be pruned. As it is necessary to observe more than qd examples in the subleaves to be able to apply the test, nodes are prevented from being pruned using the Chow test when $N_{ln_j} \leq qd$. Internal nodes having a negative error reduction or failing the test (p-value $> \alpha_{Prune}$) are pruned from the tree.

6.3.4 Guarded Incremental Pruning Algorithm

In the incremental setting, the models and statistics of the decision tree are constantly updated by each example traversing the tree. To keep the decision tree in a constantly pruned state, it is instantly pruned after each example. The pseudocode showing the incremental induction process extended by this pruning approach is given in Algorithm 7. All ILMT specific properties (refer Section 6.1) are kept unchanged. The models are learned and updated the original ILMT way, and the split statistics are also the original ILMT ones. Each example e_i traverses the tree from the root to a leaf ln. Each internal node model $in.m$ (ILMT specific) on the path is first tested with the example e_i (line 2 in Algorithm 7). The corresponding prediction error is added to the prequential error ($in.m.RSS$, line 3) and the count of observed examples is increased ($in.m.N$, line 4). Then the ILMT specific model $in.m$ is updated with example e_i (line 6). Based on the splitting decision sd_j in the node, the example is further passed to the left or right child (lines 7 to 11). Once reaching the leaf ln, the model statistics (lines 15 and 16), the model (line 18) and the ILMT specific split statistic $ln.spst$ (line 19) are updated using e_i. Subsequently, the leaf is tested for splitting. If to split ln using the most promising splitting decision sd_j (not yet tried for this node, $sd_j \notin ln.RspIL$) from $ln.spst$ is beneficial (line 21), ln is transformed into the internal node in (line 22) and two new leaves are attached. The chosen splitting decision sd_j is added to the resplit ignore list $in.RspIL$ (line 23) and the model is passed from the new internal node in to the leaves $in.left$ and $in.right$ (line 24). Additionally, $m.RSS$, $m.N$, $spst$ and $RspIL$ are reset in the new leaves (lines 25 to 28) as a new subspace is explored and specific statistics will be accumulated. Finally, the tree is pruned (lines 29 and 31). As only the prequential errors on the path have been changed, it is sufficient to test all nodes on the path for pruning. The concept of reduced error pruning [108] is adapted to the regression problem and calculated on the stored prequential error $m.RSS$ to prune the path. Pseudocode for path pruning is shown in

Algorithm 7 IncrementalInduction(Node n, Example $e_i < x_{ij}, ..., x_{ij}, y_i >$)

```
 1: if $n$ type of internal node then
 2:     predict $\hat{y}_i$ using $e_i$ with $in.m$
 3:     update $in.m.RSS$ with $y_i$, $\hat{y}_i$
 4:     increment $in.m.N$
 5:     // update ILMT specific model (e.g. FIMT variant: perceptron up-
        date)
 6:     update model $in.m$ with $e_i$
 7:     if $in.sd_j(x_{ij})$ then
 8:         IncrementalInduction($in.left$, $e_i$)
 9:     else
10:         IncrementalInduction($in.right$, $e_i$)
11:     end if
12: else
13:     // $n$ is a leaf node
14:     predict $\hat{y}_i$ using $e_i$ with $ln.m$
15:     update $ln.m.RSS$ with $y_i$, $\hat{y}_i$
16:     increment $ln.m.N$
17:     // update ILMT specific model (e.g. FIMT variant: perceptron up-
        date)
18:     update model $ln.m$ with $e_i$
19:     update $ln.spst$ with $e_i$
20:     // test for split
21:     if split $ln$ preferable using a $sd_j \notin ln.RspIL$ then
22:         $ln \to in$
23:         add $ln.spst.sd_j$ to $in.RspIL$
24:         $in.left.m \leftarrow in.m$, $in.right.m \leftarrow in.m$
25:         $in.left.m.RSS := 0$, $in.left.m.N := 0$
26:         $in.left.spst := \{\}$, $in.left.RspIL := \{\}$
27:         $in.right.m.RSS := 0$, $in.right.m.N := 0$
28:         $in.right.spst := \{\}$, $in.right.RspIL := \{\}$
29:         PathPruning($in.right$)
30:     else
31:         PathPruning($ln$)
32:     end if
33: end if
```

Algorithm 8. The path is tested from the leaf node ln up to the root to find the most harmful internal node. This is the one with the highest negative *ErrRed* or, if there is no such node, the one with the highest p-value greater than α_{prune}. Starting with the leaf node ln the example falls into, the algorithm moves up and at first the direct parent node $in_{current}$ is tested for pruning (line 3 in Algorithm 8). The node $in_{current}$ is then tested whether

Algorithm 8 PathPruning(Leaf node *ln*)

```
1: $in_{Prune} := NULL$
2: $harmScore := 0$
3: $in_{current} := ln.parent$
4: // test all nodes on the path from leaf to root
5: while $in_{current} \neq NULL$ do
6:   if not ActivePruneGuard($in_{current}$, $\xi$) then
7:     if $ErrRed(in_{current}) \leq 0$ then
8:       if $1 - ErrRed(in_{current}) \geq harmScore$ then
9:         // new most harmful node
10:        $in_{Prune} := in_{current}$
11:        $harmScore := 1 - ErrRed(in_{current})$
12:      end if
13:    else if $N_{ln_j} > qd$ then
14:      $p$-value := $ChowTest(in_{current})$
15:      if $p$-value $> \alpha_{Prune}$ then
16:        if $p$-value $\geq harmScore$ then
17:          // new most harmful node
18:          $in_{Prune} := in_{current}$
19:          $harmScore := p$-value
20:        end if
21:      end if
22:    end if
23:  end if
24:  $in_{current} := in_{current}.parent$
25: end while
26: if $in_{Prune} \neq NULL$ then
27:   // prune most harmful subtree
28:   $in_{Prune} \rightarrow ln$ // $m.RSS$, $m.N$ and $RspIL$ are maintained
29:   initialize $ln.spst$
30:   PathPruning($ln$)
31: end if
```

all q subleaves have diverged far enough (line 6). If so, the prune guard is deactivated, and the node can be tested for pruning (lines 7 to 22). Pruning is mandated, if the error reduction value is negative (line 7) or if enough ($\geq qd$) examples have been seen in the subleaves (line 13), and the positive error reduction is not significant (p-value of the Chow test is higher than α_{prune}, lines 14 and 15). As the most harmful node on the path is searched, it is tested if the harmfulness of the current node is the highest seen so far (line 8 and 16) and if so, the node is stored as the one to be pruned (lines 10 to 11 and 18 to 19). Nodes with negative error reductions are preferred over nodes with insignificant positive error reductions ($1 - ErrRed(in_{current})$ in lines 8 and 11). Using this approach, each node on the path to the root is tested. Once the most harmful node is found after traversing the path (line 26), the whole subtree is pruned from the tree by transforming the found node to a leaf node ln (line 28). The model m, including $m.RSS$, $m.N$ and the resplit ignore list $RspIL$, are inherited as the subspace for the node remains the same and the model continues its adaptation process (line 28). The split statistic *spst* is initialized (line 29) as it was no part of an internal node. As the new leaf ln changes the $ErrRed$ and Chow test for all parent nodes, all of them are again considered for pruning (line 30). This is repeated as long as there are harmful nodes on the path. Performing the pruning approach after each example is time consuming, if all the necessary information has to be collected from all subleaves, each time a node is tested for pruning. Fortunately, the prune guard as well as the pruning decision can be calculated without the need to repeatedly receive information from the subleaves. The statistics over all subleaves, necessary for the pruning decision, can be updated incrementally (RSS_{ln_j}, N_{ln_j} and q). It is easy to optimize the algorithm with respect to processing speed by storing the accumulated statistics over all subleaves ln_j in each node in. Additionally, the divergence value for each subleaf ln_j having a divergence $< \xi$ can be stored in in. The divergence list indicates which leaves activate the prune guard and how many more examples must be observed in the specific leaves until the pruning test can be performed. All numbers in in can be easily updated after each example, when the path is traversed from the leaf ln to the root and node in is scanned for pruning. That way, the prune guard as well as the pruning decision can be calculated efficiently without having to check all subleaves again and again.

6.4 Experimental Evaluation

To evaluate GuIP, we incorporated it into FIMT (see Section 3.3.2 for a detailed description) and tested it on five massive stationary datasets. First, an explanation of the experimental setup is given. Then, the results are presented and discussed.

6.4.1 Experimental Setup

FIMT was reimplemented in Java according to the published information [69]. Due to missing detailed information about the normalization step, all examples were normalized entering the tree with the method used by FIMT-DD [70].

Parameter Settings

All runs were performed with the same parameter settings considered as useful in the original publication: $\delta = 1\text{x}10^{-6}$, $\tau = 0.001$, $N_{min} = 300$ and $\eta = 0.001$. For GuIP, α_{Prune} was set by default to 0.05. All tests were run on a computing cluster consisting of different machines. All runs were randomly distributed among the machines with equal probability. The Java environment was given 3 GB of RAM for each process.

Evaluation Method

Two major evaluation methods for incremental algorithms exist: the prequential evaluation and the holdout evaluation [14]. The first one evaluates the model each time an example is observed from the data source. This is done by using each example first to test the model and then to train it. The sequentially accumulated prediction error over the observed examples is called prequential error [31] and is used as evaluation measure. Unfortunately, the prequential error learning curve is known to be a pessimistic estimator, as it suffers from potentially large errors committed during the early phases of training. To reduce or eliminate the influence of such early prediction errors, fading factors [50] or sliding windows can be used on the prequential error. The second approach to obtain accurate error estimates over time is the holdout evaluation method. In this approach, the model is evaluated periodically using a separate holdout test set. This test set is created by fetching a batch of test examples from the data source prior to training. On stationary data sources, the concept will not change and it is therefore valid to repeatedly test the model with the same holdout test set, which could even be created independently. In our work, the holdout method was preferred, as it can indicate model performance for incremental algorithms on data streams as well as on massive data sets. In the later setting, where the incremental learner is applied to a limited massive dataset, and the model is tested once the algorithm has seen all training examples, the holdout method is the natural choice. Each repeated model test can be seen as testing the model learned on a dataset up to the size of the one so far. Five massive stationary datasets were created, also representing stationary data streams. As explained, a separate holdout test set was created upfront for each training run and repeatedly used for evaluation. The trees were evaluated after every 100,000 training examples using the holdout test set. The running time needed to test the model is not included in the final runtime results. The mean absolute error MAE is used as evaluation measure, and the results for each dataset were averaged over 10 runs.

Datasets

GuIP was evaluated on five different stationary datasets: 2DimTree, 2DimFunction, 4DimTree, Census and Airline. The 2DimTree, 2DimFunction and 4DimTree source were each used to create 10 separate training sets of 50,000,000 examples with 10 holdout test sets with a size of 1,000,000 examples. No noise was added to the target variable. The airline dataset (source) was used to create ten training sets with 39,682,046 examples with ten test sets with 1,000,000 examples each. This was done by rearranging the original dataset to avoid intrinsic concept drift. The census source was used to create ten test sets by randomly choosing 500,000 examples from the whole set without replacement. The respective remainders of the set were used as the basis for generating each training set by repeated oversampling. Ten such census training sets were created, each comprising 50,000,000 examples.

6.4.2 Results

All datasets were used to evaluate the normal FIMT algorithm as well as the FIMT algorithm extended by GuIP. The effect of the prune guard parameter ξ is tested over a variety of values (299, 300, 500, 700, 1000, 2000, 3000, 4000, 8000, 16000, 24000, 50000 and 100000[1]). In the following, the influence of the prune guard parameter on the results is analyzed first, followed by an interpretation of the final results on all examples. At last, an outlook on the behavior of FIMT and GuIP using even more examples is given.

Influence of the ξ Parameter

Over all datasets, a similar effect is observed. It is exemplarily shown for four parameter settings in Figure 6.2 for the 2DimTree dataset and in Figure 6.3 the airline dataset. Measuring the *tree size*, the *memory usage*, and the *runtime* over the learning period, a clear relationship to ξ can be observed. Increasing ξ produces larger trees consuming more memory and, consequently, the runtime increases as well. While there are exceptions possible for low-dimensional and simpler datasets concerning the runtime (see Figure 6.2d), the relationship becomes more and more evident for increasing dimensionality and function complexity (see Figures 6.3a, 6.3b, and 6.3d).

Focusing on the *prediction error (MAE)* development, the relationship to ξ seems to be more complex. For the 2DimTree dataset, the MAE first decreases with increasing ξ, but rises again for higher values (cf. Figure 6.2c). On the other hand, for the airline dataset, it constantly decreases and finally stabilizes on one value (cf. Figure 6.3c). To obtain deeper insights into the

[1]As each leaf is tested for splitting every observed 300 examples (N_{min}) in the leaf, ξ=299 assures that the split at the last level can also be tested for pruning, before the child leaves could be split. For all higher ξ values, this split is prevented from pruning by the prune guard.

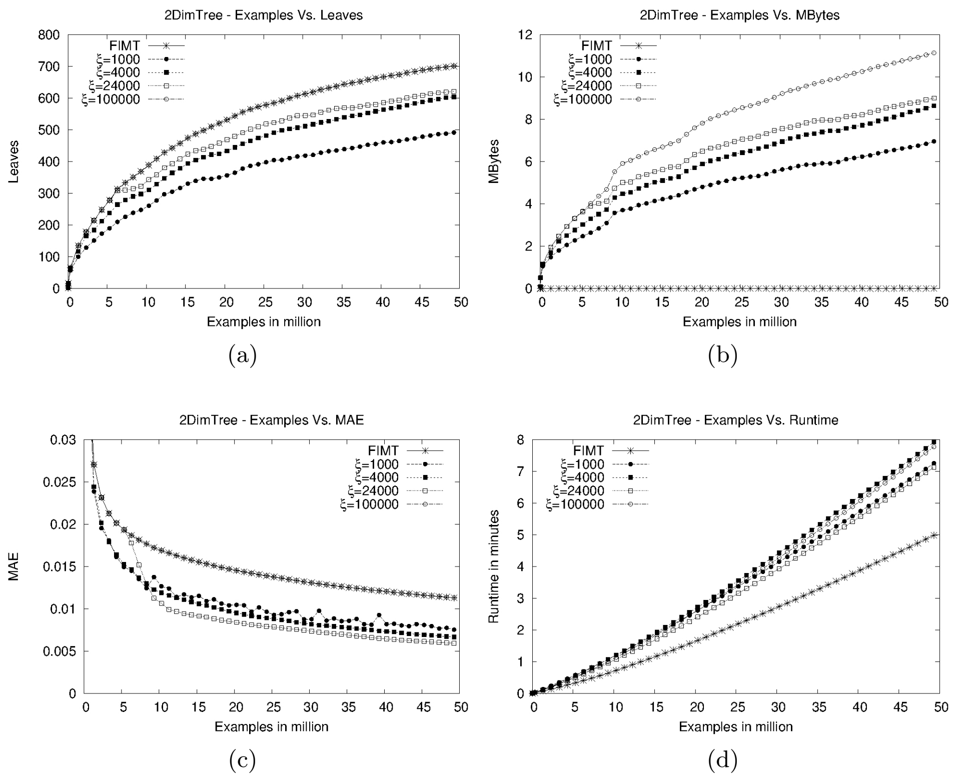
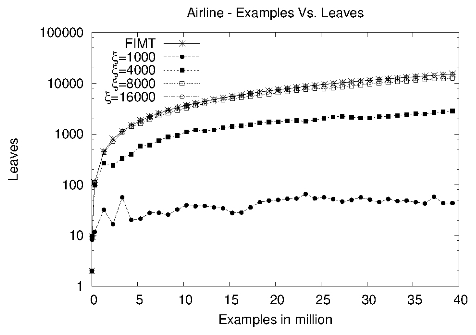


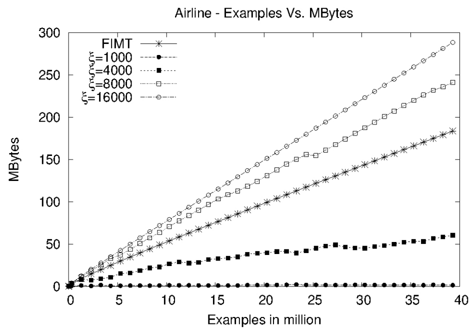
Figure 6.2: Learning curves for 2DimTree over tree size, memory usage, MAE and runtime

relation between MAE and ξ, the final mean results over all 10 runs for all ξ settings over all datasets can be found in the Tables 6.5 and 6.6 (on pages 104 and 105). These results show the mean value with standard deviations for the MAE, the runtime in minutes, the number of leaves in the tree, and the memory consumption in mega bytes over the 10 runs. The results are compared to FIMT and improvements are displayed in bold. All differences were tested for significance using the Wilcoxon Signed-Rank Test for paired samples [141] and non significant differences (p-value > 0.05) are highlighted (●). The derived relationship between the final MAE and ξ is schematically shown in Figure 6.4. For datasets with a high function complexity, representing a highly non-linear problem (e.g., airline dataset), a constant decrease of the MAE with increasing ξ can be observed, finally stagnating on the value also achieved by FIMT. For datasets with a more piecewise linear function, the MAE is decreasing until a minimum is found and increases again for higher values of ξ[2]. Again, the MAE stagnates on the value also

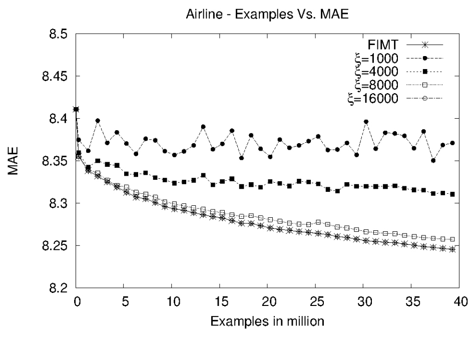
[2]Even though this phenomenon resembles overfitting, this is not the case, as it is achieved over different settings of ξ. Overfitting is generally not found on any dataset,



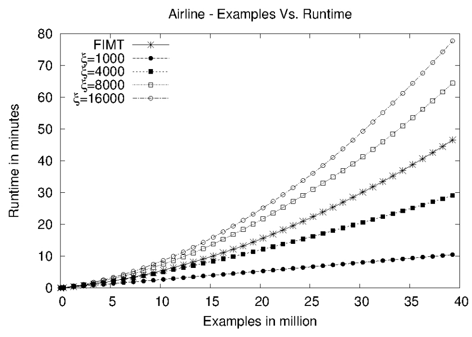
(a)



(b)



(c)



(d)

Figure 6.3: Learning curves for Airline over tree size, memory usage, MAE and runtime

achieved by FIMT for high values of ξ. For a specific range of ξ, the MAE is lower than the one achieved by FIMT. With increasing piecewise linearity in the datasets (census $\rightarrow$ 2DimFunction $\rightarrow$ 4DimTree $\rightarrow$ 2DimTree), this range expands and the MAE improvement at the minimum becomes more prominent. The ξ value with the minimum MAE depends on the dataset. The particular ξ with minimum MAE can be explained by the concept and functioning of the prune guard. It defends a subtree from being pruned, if its subleaves have not diverged far enough from its root node. Starting from a single node, the subtree is expanding by seeing more and more examples. Now, depending on the value of the ξ parameter, the subtree is sooner or later tested for a significant prediction gain. For small values of ξ, the subtree is tested earlier, and fast significant reductions in the prediction error are needed or the subtree will be pruned. In this way, for too small values of ξ, temporarily non-significant beneficial subtrees are not accepted, which could be preferable for the prediction accuracy in the long run. By increas-

as the MAE learning curves are not increasing with more examples (cf. Figures 6.2c and 6.3c).

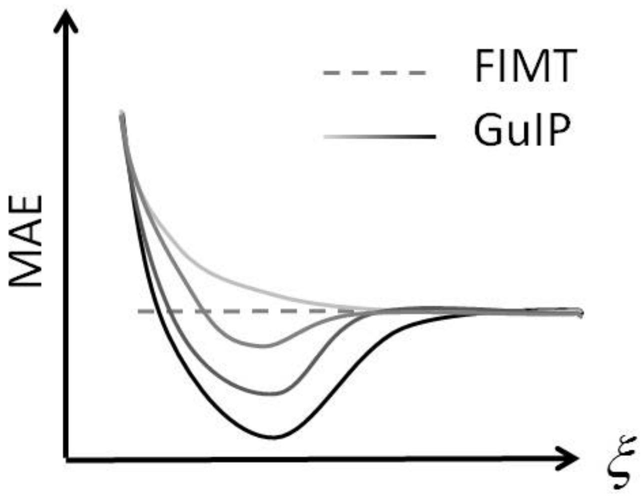


Figure 6.4: Relationship between MAE and ξ for different function complexities

ing ξ, the guarded subtrees are getting bigger, consisting of an increasing number of levels. By that, subtrees with a possible non-significant prediction gain in early stages, are guarded from being pruned and can result in a significant, beneficial subtree. This is supported by the results from the Tables 6.5 and 6.6 (on pages 104 and 105), as the MAE improves with increasing ξ. Datasets with a highly non-linear function even benefit from high values of the ξ parameter. For these kind of datasets, pruning itself is harmful as the calculated piecewise linear approximation still improves with every further split and tree growth. Consequently, the final MAE benefits from strong prune guards, resulting in larger subtrees. Due to the finer piecewise linear approximation of these subtrees, the prediction gain will become more pronounced, resulting in less pruning. Ultimately, no pruning will occur, and the same results as the ones from FIMT are obtained. For datasets with a more piecewise linear function, setting the prune guard to too high values would harm the final MAE. Although the guarded subtrees will usually result in a significant prediction gain, mostly suboptimal subtrees will be found. In contrast to high non-linear functions, a better prediction can be reached here by guiding the subtree to a more optimal tree. This can be achieved by adjusting the ξ parameter in such a way that the subtree still has the chance to grow and overcome first non-significant prediction gains, but is still guided by pruning to the optimal subtree.

Final Results

As mentioned above, the results in the Tables 6.5 and 6.6 (on pages 104 and 105) display the final mean results after observing all examples from the stationary data sources. To be able to evaluate the MAE differences between the FIMT and GuIP approaches properly, a FIMT tree restricted to only one node (resulting in a perceptron algorithm) was used as baseline algorithm and also run on all datasets (cf. Tables 6.5 and 6.6 *Perceptron*). The results should be interpreted in the order of increasing data dimensionality and linear function complexity. It can be observed that the runtime is increased using the pruning approach on the datasets with a very small dimensionality (≤ 2). This is due to the small number of dimensions combined with the pruning overhead (pruning after each example, updating the models and statistics in each node). This overhead is further boosted by the relatively large absolute tree sizes and the small differences in size between the pruned trees and the unpruned trees. Nevertheless, the prediction error is significantly improved for the 2DimTree dataset by 48% with the best ξ setting ($\xi = 24,000$) using a 11% smaller tree. If tree size and memory consumption is in focus, the MAE can still be significantly improved (by 35%) using a 30% smaller tree needing 30% less memory ($\xi = 1,000$). Increasing the dimensionality, as done on the 4DimTree dataset, delivers more accurate models, smaller trees and a runtime advantage. An error improvement of 40% with a 55% smaller tree (52% less memory), which was learned in only 61% of the time, can be found with the best setting ($\xi = 1,000$). Choosing a highly non-linear function with very small dimensionality, as done on the 2DimFunction dataset, indicates that pruning is costly for this function. For the best MAE setting, a significant MAE improvement can be achieved at the cost of a higher runtime, needing more memory ($\xi = 50,000$). Nevertheless, the tree size could be decreased by 31% ($\xi = 1,000$) with only a negligible increase in prediction error. Real-world datasets, including census and airline, are often very high-dimensional. Applying the different pruning settings on the census dataset resulted in a slightly better (but not significant) improvement of the MAE with a 28% smaller tree. If only a marginal MAE increase is acceptable, the runtime can, for example, be decreased by 37%, building a 95% smaller tree, needing 90% less memory ($\xi = 4,000$). These percentages can still be improved by lowering the ξ parameter. The airline dataset is the one with the most complex non-linear function. Applying pruning on the dataset is not beneficial at all for the MAE reduction. The best MAE is achieved if pruning is disabled. Nevertheless, when the prediction accuracy is not in focus, but runtime limitations raise the need for fast learners, GuIP can speed up the process by only marginally increasing the MAE. For $\xi = 4,000$, a runtime improvement of 38% with a tree size decrease of 81% consuming 67% less memory can be found. This is done at the expense of increasing the MAE by only 0.07, representing a deterioration of the prediction by only 4.2 seconds for a target variable spanning 189 minutes. In application domains where marginally worse predictions are acceptable, the tree size, memory usage as well as the processing speed can benefit considerably from GuIP.

Outlook

To enable further insights into the advantage of GuIP over time, learning curves for all datasets over the difference between the values of FIMT and FIMT with GuIP (FIMT - GuIP ξ=4000) are analyzed (see Figure 6.5). Figure 6.5b shows that the MAE difference between the two approaches reaches a stable state (for the airline dataset beginning at 30 million examples), while the tree size difference increases steadily (see Figure 6.5a, please notice the log-scale) and, consequently, the runtime difference increases as well (refer Figure 6.5c). This effect is strongly boosted with increasing data dimensionality and complexity of the target function. This shows that the more examples are processed, the more evident the speed advantage enjoyed by the ILMT using GuIP becomes, while predictive performance remains competitive. The small two-dimensional datasets are an exception concerning the runtime, because of the above-mentioned pruning overhead.

6.5 GuIP Application Scenarios

GuIP is an extension of ILMTs for stationary data sources. These data sources could either be stationary data streams or massive datasets. The usage of GuIP in these application domains is further motivated in this section.

6.5.1 Stationary Data Streams

The first application scenario of GuIP lies in the area of data streams. If the data stream speed is slower than the algorithm intrinsic maximal example processing speed, every example from the data stream can be processed and predictions can be made. If, however, the processing speed is too slow relative to the stream speed, most of the approaches have to leave out data which could be useful or even essential for the learning process (e.g., methods based on load shedding or sampling [84, 77]). Special frameworks are being developed to handle every example from even very high speed data streams overwhelming the learning algorithm [62]. Nevertheless, additional memory and CPU power is permanently needed. Using GuIP on ILMTs gives the chance to adopt the algorithm processing speed directly to the data stream speed by adjusting ξ. Doing so, the maximal prediction accuracy can be gained by still processing all examples. To find the ξ value best suitable for the data stream speed, several parameter settings can be used in parallel, and all but one with the best prediction accuracy still covering the data stream speed can be removed. For the cases of altering data streams, again several ILMTs using GuIP with different ξ parameter can be used. Depending on the actual data stream speed, the optimal model can be picked. After a certain number of examples, all ILMTs with a ξ value higher than the current best one (lowest test MAE / minimum) can be removed, as they have equal or even worse MAE and longer runtime.

6.5.2 Massive Datasets

To learn from massive datasets, batch algorithms are still the state of the art. These kind of algorithms have to see the whole dataset at once at training time. This is usually done by loading and processing all examples in main memory. For the cases where main memory is too small to store all examples, alternative approaches have been developed. A widely used approach is, e.g., sampling, where a representative subset from the whole dataset is chosen, and then fed into a complex learning algorithm. A recent article by P. Domingos [34], however, has argued that the fastest way to an improved accuracy is to use more data in a simpler algorithm instead of a limited amount of data in a complex algorithm. Complex learning algorithms take much longer to be learned, consume more CPU time and especially require more cycles of parameter optimization and human involvement. This is due to their harder usage: Internals are more opaque and more tuning needs to be done to achieve good results. To evaluate the prediction gain with more examples, we further compare the performance of the online learning approaches with the performance of a batch learner using sampling. This is done by using the Weka-Workbench Version 3.7.5, which is also implemented in JAVA and thus ensures trustworthy runtime comparisons. For each process, the Java environment received once again 3 GB of RAM. As the M5 algorithm is the batch version of the FIMT algorithm, the implementation in Weka (M5P [140]) is used with default parameters. Smoothing is deactivated, as it is also not used in FIMT and pruning is activated. For the sampling method, the reservoir sampling algorithm "R" [136] was chosen. As before, M5P & R is used on each of the 10 training sets for each dataset and tested on the corresponding test set. The results are again averaged. Table 6.2 shows the maximal amount of examples that can be used to train the model using 3 GB of RAM. Depending on the complexity of the dataset, the amount of examples is between 5.8 million and 0.8 million examples. While the batch learner (M5P & R) can only use a very small subsample of the whole dataset consuming 3 GB of RAM, the online learner (FIMT & GuIP) is able to process all examples with a much lower memory consumption (cf. Table 6.3). Comparing the batch algorithm results in Table 6.4 with the online learning results in the Table 6.5 and 6.6 shows an advantage of the batch learner for the datasets with a simple piecewise linear

Table 6.2: Maximal amount of processable training examples for each dataset.

Dataset	**Examples in mio.**
2DimTree	5.8
2DimFunction	4.9
4DimTree	4.3
airline	1.4
census	0.8

Table 6.3: Maximal model memory usage of the online learner (FIMT with GuIP).

Dataset	**MBytes**
2DimTree	11.19 (0.34)
2DimFunction	3.96 (0.14)
4DimTree	261.59 (7.97)
airline	302.49 (2.70)
census	118.44 (2.73)

function. The 2DimTree dataset is a dataset with a very simple piecewise linear function, which favors linear model trees. Consequently, only a small subsample is enough for the batch learner to build a well-performing model for the function. The same is true for the 4DimTree dataset. But because of the increased complexity in the 4DimTree dataset, the performance gap between the online and the batch approach becomes smaller. In fact, with a suitable value of ξ (e.g. $\xi = 1,000$), the MAE for the online approach is even lower than for the batch learner. The advantage of the online learning approach can be seen with even more complex, non-linear datasets. Beginning with the airline dataset, the online learner could process 48.6 million examples more in only 60% ($\xi = 4,000$) of the time the batch algorithm needed, showing an improved prediction accuracy. For higher values of ξ, the online learner even further improves its prediction accuracy, but needs more runtime than the batch algorithm. For the census dataset, a marginally lower prediction accuracy is achieved by processing even 49.2 million more examples in only 56% ($\xi = 4,000$) of the time, or a lower MAE can be obtained in 86% of the time ($\xi = 8,000$). The biggest improvement in the prediction error can be observed on the 2DimFunction dataset, where the online learner has a 67% lower MAE by processing 45.1 million more examples using only 70% of the time ($\xi = 50,000$). As can be seen in Figure 6.2c and 6.3c, the online learner with the appropriate parameter setting improves its prediction accuracy even further with every new example. Consequently, the

Table 6.4: Performance of M5P & R with pruning using the maximal dataset specific training examples.

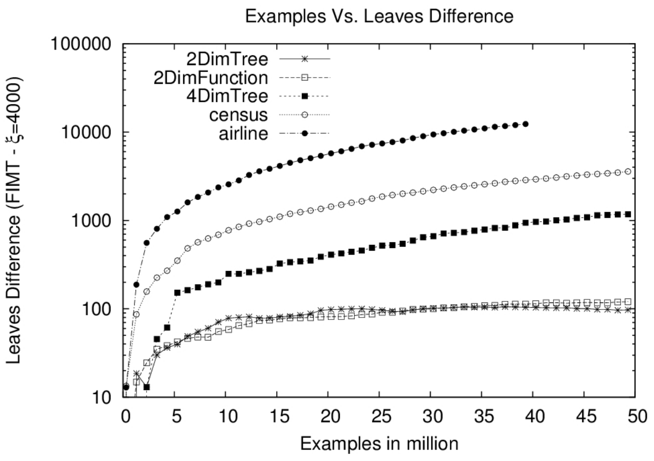
Dataset	MAE	Runtime in min.	Leaves
2DimTree	0.0007 (0.0000)	6.05 (0.42)	21.4 (1.1)
2DimFunction	0.0067 (0.0002)	7.11 (1.02)	322.1 (10.8)
4DimTree	0.1894 (0.0173)	11.78 (1.36)	234.3 (39.73)
airline	8.3306 (0.0101)	49.03 (6.55)	251.7 (34.9)
census	0.3154 (0.0016)	37.48 (4.52)	142.6 (38.7)

online learner can steadily improve its prediction performance with larger datasets, while the batch learner's prediction performance is limited by the examples fitting into main memory.
These results show the advantage of GuIP and FIMT over its batch equivalents, when a suitable ξ value is chosen. Finding this value may appear as a challenge at first glance. But only two scenarios are possible in practice when working with datasets. Sufficient time is available or time is a limiting factor that can be very strong. When sufficient time is available, several ξ parameters can be tested to approach the MAE minimum for the best model. The minimum can be identified when the MAE is rising again with increasing ξ values. But in most applications, time is a strong limiting factor. Results have to be gained within a very short period of time by possibly running the algorithm several times to obtain, e.g., the best parameter settings. While long runtimes would hinder this evaluation process or even make it impossible [34], GuIP reduces the runtime substantially by choosing a small ξ value. Once the proper setting for the algorithm is found, and first useful results are available, ξ can be increased stepwise to obtain further MAE improvements, if needed.

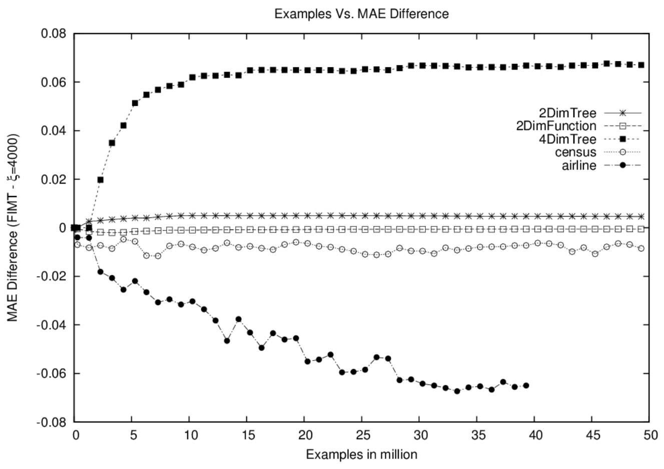
6.6 Conclusion

The guarded incremental pruning approach GuIP for stationary data sources was presented. It is an extension of incremental linear model trees with approximate lookahead in general and was exemplarily integrated into the FIMT algorithm. Results on five massive datasets showed that the prediction accuracy, tree size, memory consumption and consequently, the example processing speed are influenced by the prune guard parameter ξ. By adjusting ξ, a prediction accuracy gain can be achieved depending on the degree of dataset complexity. For less complex datasets, this gain can be achieved along with producing significantly smaller trees in a fraction of the time. For more complex datasets, more time is needed to achieve a better or equal accuracy. When runtime is a limiting factor, decreasing ξ can immensely speed up the learning process by a significant reduction of tree size. Moreover, for less complex datasets, a smaller ξ value may still result in a better prediction accuracy, while for more complex datasets, a marginal decrease of prediction accuracy can occur. Consequently, depending on the requirements, if processing speed or prediction accuracy is more important, the tree can be tuned accordingly. The tree size and processing speed advantage becomes even more pronounced given more and more examples without any prediction disadvantage. Additionally, the advantage over the equivalent batch algorithm has been shown on complex massive datasets. Improved prediction results are obtained in only a fraction of the time needed by the batch algorithm. Moreover, additional examples still improve the prediction accuracy of the online algorithm, which leads to a more marked advantage for even larger datasets.

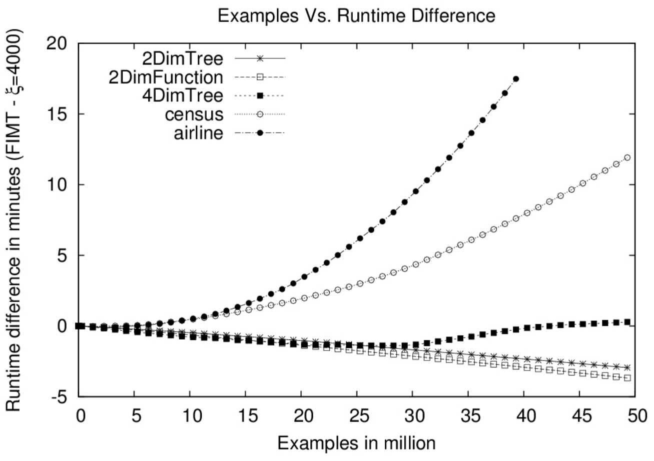
Developing a more sophisticated adaptive method for choosing the optimal value for ξ is a direction for future work. Moreover, it would be interesting to apply GuIP to the domain of evolving data streams, to evaluate its ability to detect concept drift. Additionally, it may be interesting to investigate periodical pruning (pruning after processing multiple examples, instead of pruning after every single example), to further increase the processing speed. Finally, it would interesting to see if any of the presented techniques could be useful for the development of online quantile regression trees.



(a)



(b)



(c)

Figure 6.5: Learning curves for tree size, MAE and runtime over all 39.68 million examples in the airline dataset and 50 million examples in the remaining datasets. The difference between the value of FIMT and ξ=4000 (FIMT - ξ=4000) is plotted for each evaluation measure.

Table 6.5: Results on the artificial datasets after 50 million examples presenting mean (stdev). Non-significant differences to FIMT using the Wilcoxon Signed-Rank Test are marked with • (p-value > 0.05).

	2DimTree			
	MAE	Runtime in min.	Leaves	MBytes
Perceptron	1.1025 (0.0021)	2.31 (0.12)	1 (0)	0.20 (0)
FIMT	0.0113 (0.0011)	5.07 (0.54)	704.4 (24.0)	10.08 (0.32)
$\xi =$				
299	0.0311 (0.0262)	8.28 (1.44)	**226.9** (118.7)	**3.31** (1.59)
300	0.0227 (0.0251) •	7.61 (1.13)	**288.5** (169.3)	**4.33** (2.34)
500	0.0148 (0.0064) •	8.40 (1.24)	**325.6** (129.9)	**4.46** (1.81)
700	**0.0081** (0.0041) •	7.51 (0.90)	**435.9** (87.2)	**6.07** (1.40)
1000	**0.0074** (0.0023)	7.38 (0.97)	**496.2** (94.4)	**7.01** (1.55)
2000	**0.0068** (0.0008)	7.72 (1.22)	**556.0** (84.1)	**7.83** (1.35)
3000	**0.0065** (0.0007)	7.83 (0.89)	**590.9** (43.3)	**8.38** (0.71)
4000	**0.0066** (0.0006)	8.07 (0.83)	**606.0** (28.1)	**8.68** (0.39)
8000	**0.0067** (0.0007)	7.24 (0.81)	**613.9** (26.85)	**8.77** (0.31)
16000	**0.0067** (0.0006)	7.66 (0.25)	**622.6** (24.36)	**9.00** (0.35)
24000	**0.0059** (0.0011)	7.26 (0.21))	**624.4** (29.94)	**9.05** (0.38)
50000	**0.0066** (0.0010)	7.43 (0.24)	**642.1** (26.96)	**9.35** (0.45)
100000	0.0113 (0.0011) •	7.92 (0.41)	**704.2** (23.99) •	11.19 (0.34)

	2DimFunction			
	MAE	Runtime in min.	Leaves	MBytes
Perceptron	0.34603 (0.00179)	1.99 (0.08)	1 (0)	0.04 (0)
FIMT	0.00221 (0.00026)	2.55 (0.29)	1083.9 (47.5)	2.43 (0.07)
$\xi =$				
299	0.03496 (0.01305)	4.84 (0.46)	**381.1** (49.3)	**1.49** (0.17)
300	0.03145 (0.00879)	4.76 (0.35)	**403.7** (42.6)	**1.57** (0.14)
500	0.02415 (0.00579)	5.27 (0.49)	**489.2** (89.9)	**1.80** (0.28)
700	0.02042 (0.00645)	5.52 (0.74)	**554.1** (73.8)	**2.00** (0.22)
1000	0.00609 (0.00095)	6.18 (0.56)	**752.0** (53.4)	2.65 (0.17)
2000	0.00343 (0.00055)	6.07 (0.64)	**885.8** (52.9)	3.06 (0.13)
3000	0.00294 (0.00050)	6.07 (0.70)	**927.1** (49.8)	3.17 (0.12)
4000	0.00271 (0.00035)	6.28 (0.69)	**963.6** (61.9)	3.27 (0.14)
8000	0.00244 (0.00032)	6.24 (0.64)	**1005.0** (58.7)	3.38 (0.15)
16000	0.00235 (0.00026)	5.69 (0.23)	**1025.5** (56.5)	3.46 (0.13)
24000	0.00223 (0.00027) •	5.34 (0.14)	**1060.7** (42.9)	3.59 (0.10)
50000	**0.00219** (0.00026)	4.99 (0.24)	**1081.9** (48.8) •	3.74 (0.13)
100000	0.00221 (0.00025) •	5.06 (0.12)	1083.8 (47.41) •	3.96 (0.14)

	4DimTree			
	MAE	Runtime in min.	Leaves	MBytes
Perceptron	9.5151 (0.0895)	3.87 (0.13)	1 (0)	0.40 (0)
FIMT	0.2804 (0.0198)	82.05 (9.09)	10590.6 (331.0)	238.61
$\xi =$				
299	9.5151 (0.0895)	**27.74** (7.39)	**1** (0)	**0.51** (0)
300	9.5151 (0.0895)	**31.51** (10.22)	**1** (0)	**0.51** (0)
500	8.4211 (0.8923)	**7.32** (0.56)	**12.6** (6.6)	**0.54** (0.29)
700	0.3823 (0.0488)	**28.04** (2.89)	**1681.8** (287.4)	**45.21** (7.63)
1000	**0.1681** (0.0338)	**50.03** (8.66)	**4784** (1061.4)	**115.64** (19.76)
2000	**0.1840** (0.0266)	**67.97** (6.54)	**8079** (386.5)	**184.72** (8.33)
3000	**0.2007** (0.0168)	**72.68** (8.28)	**8975.4** (366.9)	**203.43** (9.6)
4000	**0.2127** (0.0165)	**81.71** (9.57) •	**9370.2** (305.7)	**212.42** (7.90)
8000	**0.2447** (0.0201)	85.50 (8.74) •	**10342** (346.07)	**238.24** (7.67) •
16000	**0.2632** (0.0228)	95.00 (3.78)	**10548.9** (328.6)	250.6 (8.21)
24000	**0.2729** (0.0233)	94.08 (4.87)	**10577.2** (329.54)	257.35 (7.92)
50000	0.2804 (0.0198) •	97.36 (5.69)	10590.6 (331.0) •	261.52 (7.98)
100000	0.2804 (0.0198) •	95.29 (8.41)	10590.6 (331.0) •	261.59 (7.97)

Table 6.6: Results on the real-world datasets after 50 million and 39.68 million (airline) examples presenting mean (stdev). Non-significant differences to FIMT using the Wilcoxon Signed-Rank Test are marked with • (p-value > 0.05).

	airline			
	MAE	Runtime in min.	Leaves	MBytes
Perceptron	8.3883 (0.0101)	9.28 (0.23)	1 (0)	0.25 (0)
FIMT	8.2451 (0.0098)	47.43 (5.47)	15351.2 (103.1)	185.49 (2.03)
$\xi =$				
299	8.3879 (0.0084)	**10.43** (1.01)	**5.1** (1.3)	**0.21** (0.07)
300	8.3872 (0.0103)	**10.50** (0.87)	**6.0** (3.1)	**0.32** (0.11)
500	8.3823 (0.0177)	**10.91** (0.91)	**19.9** (11.4)	**0.74** (0.41)
700	8.3754 (0.0133)	**10.64** (0.86)	**51.7** (25.5)	**1.74** (0.77)
1000	8.3728 (0.0189)	**10.57** (0.78)	**52.5** (26.8)	**1.87** (0.87)
2000	8.3566 (0.0110)	**13.66** (1.72)	**325.1** (136.0)	**8.88** (3.16)
3000	8.3443 (0.0098)	**18.08** (1.66)	**797.6** (363.8)	**19.80** (8.36)
4000	8.3126 (0.0145)	**29.51** (4.12)	**2888** (1025.0)	**61.32** (20.64)
8000	8.2566 (0.0095)	65.62 (7.13)	**12803.6** (1710.6)	243.85 (29.74)
16000	8.2451 (0.0097) •	79.13 (2.82)	**15338.2** (116.9) •	291.47 (2.95)
24000	8.2451 (0.0098) •	79.99 (2.45)	15351.2 (103.1) •	294.04 (2.67)
50000	8.2451 (0.0098) •	85.37 (1.33)	15351.2 (103.1) •	299.93 (2.63)
100000	8.2451 (0.0098) •	84.40 (2.10)	15351.2 (103.1) •	302.49 (2.70)

	census			
	MAE	Runtime in min.	Leaves	MBytes
Perceptron	0.40431 (0.0029)	18.80 (0.39)	1 (0)	0.05 (0)
FIMT	0.3135 (0.0010)	33.13 (0.91)	3847.9 (89.5)	69.00 (1.53)
$\xi =$				
299	0.3485 (0.0019)	**18.10** (1.14)	**2.7** (0.7)	**0.14** (0.03)
300	0.3485 (0.0018)	**17.58** (0.33)	**2.5** (0.7)	**0.13** (0.03)
500	0.3483 (0.0017)	**17.89** (0.82)	**4.5** (2.0)	**0.20** (0.07)
700	0.3464 (0.0035)	**18.65** (1.19)	**4.1** (2.6)	**0.19** (0.09)
1000	0.3455 (0.0032)	**18.18** (0.75)	**7.9** (3.0)	**0.33** (0.12)
2000	0.3382 (0.0095)	**18.79** (1.22)	**25.8** (14.2)	**0.98** (0.52)
3000	0.3280 (0.0069)	**19.39** (1.17)	**91.8** (102.2)	**3.25** (3.45)
4000	0.3241 (0.0084)	**20.88** (1.20)	**201.1** (145.9)	**6.97** (4.91)
8000	0.3140 (0.0012)	**32.37** (5.23) •	**1217.9** (377.70)	**38.96** (11.48)
16000	**0.3133** (0.0009) •	48.50 (6.96)	**2787.9** (688.03)	85.57 (19.90)
24000	0.3135 (0.0010) •	53.86 (1.59)	**3724.3** (114.0)	112.90 (3.22)
50000	0.3135 (0.0010) •	57.29 (1.68)	3847.9 (89.5) •	117.25 (2.67)
100000	0.3135 (0.0010) •	57.33 (2.12)	3847.9 (89.5) •	118.44 (2.73)

Chapter 7

Towards Real-Time Machine Learning

7.1 Introduction

Over the past few years, the amount of collected information has been increasing extremely, and new challenges are posed to classical machine learning algorithms. Web applications, social services and sensors capturing the environment with increasing quality produce a steadily growing mass of data. Next generation sequencing (NGS) technologies double their sequencing capacity of base pairs (bp) per dollar every fifth month [125], the "LifeShirt" project [23] monitors the health status of patients with body sensors, generating 200 MB over 24 h for one person, and the NASA Earth Observation System (EOS) produces 2.9 TB data per day [24], to name just a few prominent examples. Such high-speed data streams (DS) can be found in many other areas beyond science as well, like finance, web applications or telecommunications.

While the mass of data is steadily increasing and data streams constantly gain in speed, online learning algorithms used to process the data are naturally limited by their maximal instance processing speed. As the evolution of these massive data streams is much faster than the improvement of CPU power after Moore's law [100], the gap increases between available and processable data. As a consequence, not all provided instances in a stream can be used by the learning algorithm and some have to be skipped. This could be especially harmful if the skipped instances would reveal important insights to the user. To avoid skipping instances and to still enable (potential) insights, currently not processable instances could, in principle, be externally stored for later processing. However, as the data stream speed is higher than the processing speed, the algorithm is constantly challenged by the amount of data, and the amount of stored instances is constantly increasing. Besides memory usage, the time span from the arrival of the instances to their processing (response time) is steadily increasing as well. This processing delay can result in outdated information, and important events might be missed. To address these issues, this chapter introduces PAFAS (Prediction Assured Framework for Arbitrarily Fast Data Streams) [62], a framework to handle high-speed data streams that potentially go to or beyond the limits of the processing speed of the online learning algorithm. The contributions of PAFAS are:

1. All unlabeled instances in the data stream receive a prediction.
2. The prediction is given promptly after the instances' arrival time.
3. The prediction model is constantly improved (independent of the DS speed).
4. No external instance storage is needed.

The framework can be applied whenever events have to be detected as soon as possible, and no information is allowed to be missed to detect these events. We believe that concepts for the embedding of machine learning into real-world systems are required, taking into account the time it takes to make a prediction as well as the time it takes to train or refine a model. In this chapter, we discuss one such framework and present evidence from experiments with varying loads.

This chapter is organized as follows. First, related work is presented. Then, the problem setting is presented along with the proposed framework. Subsequently, the evaluation of the framework is presented in Section 7.4. The chapter closes with a discussion.

7.2 Related Work

Data stream mining has developed considerably in the past decade and attracted many researchers to adopt existing algorithms for the challenging task to process and reason about instances received at a very high speed [46]. One part addresses the adaptation of batch algorithms to cope with the data stream setting [36] by, e.g. incremental batch approaches [138]. To provide a specific environment for efficient data stream processing, data stream management systems (DSMS) have been developed. Such systems are adaptations of database management systems (DBMS) to query continuous, unbounded data streams possibly in combination with pre-stored, fixed datasets. Two well-known DSMS, AURORA [1] and STREAM [7], use their own language to query data streams. Both systems also address the problem of too fast data streams, i.e., when the system is not capable of processing all of the instances provided by the data stream. They use *load shedding* (also implemented in a system environment [26]) to select instances of the data stream that should be processed. Based on Quality-Of-Service (QoS) specifications, the system decides which instances are useful for the system to fetch and which instances can be discarded. The main idea is to select instances that will most probably lead to a good prediction. Another possibility to cope with too fast data streams is *sampling*. Sampling is a technique to represent a larger dataset by a smaller selected subset. It was frequently applied to reduce the overall processing time of data mining algorithms and to efficiently scan large datasets [128]. In the simplest case it selects a random subset from the whole data set as an input for the learner. Frequently, the purpose of this is to estimate the quality of the result [35]. Another possibility to cope with very fast data streams is to adapt the mining technique corresponding to the currently available resources. Such methods are summarized under the heading of *granularity-based techniques*. While load shedding and sampling change the input granularity of the data mining method, the output of the data mining method can also be reduced, e.g. the number of rules or clusters [45]. Then, the model that is used for classification is smaller and thus also more time-efficient, i.e., more instances can be processed in less time. This method termed Algorithm Output Granularity (AOG) can also be applied to various data mining schemes like clustering, classification or frequent set mining. Last, anytime algorithms are also often used for altering data stream speeds, as they can be interrupted anytime to return an intermediate result [123]. The more time available, the better the result has to be. Most of the presented approaches make use of a resource monitor (also called controller) that decides how an instance will be processed, depending on the current data stream speed. We will also make use of this successful concept in our work. However, none of these methods addresses the case when there are labeled and unlabeled instances in the data stream and the user expects a classification for each unlabeled instance. If one applies load shedding or sampling on such a data stream, instances may drop out of the process and no prediction would be made for them. If AOG was used in such a case, then still the data stream may be too fast for even the smallest model. This would either lead to a memory exception or long response times for such instances. Anytime algorithms need an initialization period for each instance and consequently, they can be overwhelmed by fast data streams as well. Therefore, we propose an approach that guarantees prompt prediction of each unlabeled instance by adaptation to data streams of varying speeds.

7.3 Prediction Assured Framework for Arbitrarily Fast Data Streams (PAFAS)

This section first introduces the problem setting and then specifies PAFAS, which is proposed to tackle the problem.

7.3.1 Problem Setting

A data stream $DS = \{i_1, \ldots, i_j, \ldots, i_\infty\}$ is a possibly unbounded sequence of instances $i \in \mathbb{R}^k$ observed in increasing order of index j, where each instance is observed at a specific time point t_j. Each instance $i_j =< x_{j1}, \ldots, x_{jk-1}, y_j >$ consists of attributes with known values (x_{jk}) and an attribute of interest (y_i, the target variable) with a possibly missing value. Depending on the attribute of interest, we distinguish between two types of data streams. In the first data stream type, the value for the attribute of interest is given (DS_L / labeled data stream) and in the second, the attribute value is missing (DS_U / unlabeled data stream). As the value of y_j in DS_U is important in the application domain, a model M is trained on DS_L, where y_j is known for each instance. Model M is then applied on DS_U to make a prediction $\hat{y_j}$ for y_j. For simplicity, model and learning algorithm are merged into one entity in our framework, incorporating both the representation of the model (function) and learning / adaptation functionality. Each data stream has a specific speed $\vec{v}_{DS}$, defined as the number of instances observed in the streams in a specific time interval. Furthermore, each model M has a specific instance processing speed $\vec{v}_M$, defined as the number of instances processable in a specific time interval. For high speed data streams, $\vec{v}_L >> \vec{v}_M$ and $\vec{v}_U >> \vec{v}_M$. Consequently, not all instances $i_j \in DS_L$ and $i_j \in DS_U$ can be processed by M. As the prediction of $\hat{y_j}$ for all $i_j \in DS_U$ is essential in the application domain, the task is to predict $\hat{y_j}$ as soon as possible after receiving instance i_j from DS_U. This prediction has to be made as good as possible for all $i_j \in DS_U$. Instances without a $\hat{y_j}$ prediction are not allowed in our envisaged usage.

7.3.2 Approach Specification

To address the given problem setting, we integrate a so-called *controller* into the online process (cf. Figure 7.1, center). The controller fetches the instances over a specific time interval t_f, which we refer to as *fetching interval*, from the data streams DS_L and DS_U. These instances are defined as X'_{tr} for the instances fetched from the data stream DS_L and X'_{pr} for the instances fetched from DS_U over the time interval t_f. After each fetching interval, the controller uses the instances in X'_{tr} to further train the model M and the instances in X'_{pr} to receive predictions from M. Meanwhile, new instances are collected in the new fetching interval. That way, the controller works as a buffer between the data streams, where the instances can arrive in altering time intervals, and M, where the instances are processed at a constant speed. Furthermore, the controller assures that M is only used by the instances in a time interval of t_f and that the model is then available for the next instances in X'_{tr} and X'_{pr} from the next fetching interval. As it is mandatory that all instances $i_j \in X'_{pr}$ receive a value $\hat{y}$, the controller

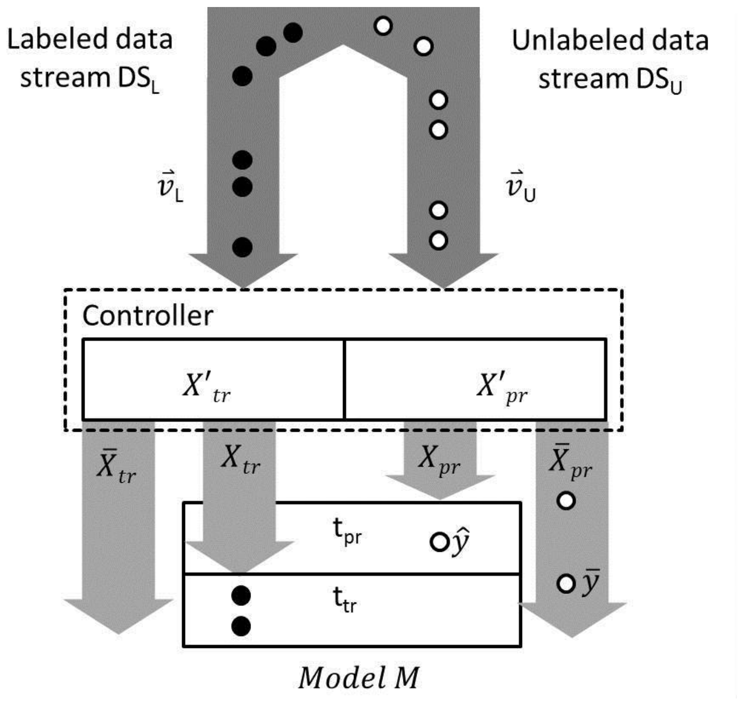


Figure 7.1: Illustration of the problem setting. Two data streams DS_U and DS_L provide instances with speed v_L and v_U. The task is to predict $\hat{y}_i$ as soon as possible for each instance from DS_U using model M. To handle very fast data streams, the controller manages the instance processing.

prefers these instances and passes them before the instances in X'_{tr} to the model M. In the case where the data stream speed is constantly higher than the instance processing speed of the model (i.e. $\vec{v}_U > \vec{v}_M$), a state is reached where not all instances from DS_U would receive a prediction. Additionally, the model M would not be further improved by the instances from DS_L, because no time is left to process these instances. That is why the time interval t_f is divided into t_{pr} and t_{tr} ($t_{pr} + t_{tr} = t_f$). t_{pr} is the maximal time given to the instances in X'_{pr} to receive a prediction $\hat{y}$ from the model M, and t_{tr} is the maximal time given to the instances in X'_{tr} to improve the model M. As DS_L and DS_U can have different speeds, the amount of instances in X'_{tr} and X'_{pr} can be highly unequal. An equal share of the time would not be suitable. Consequently, t_{pr} is calculated as the proportional number of instances in X'_{pr} over all instances in the controller. Therefore, the time given for the instances in X'_{pr} is

$$t_{pr} = \frac{X'_{pr}}{X'_{pr} + X'_{tr}}.$$

However, as $\vec{v_M}$ is normally faster for the prediction task than for the training task, there may be the case where the time for the prediction is not fully required, and a prediction for all instances in X'_{pr} is given in t'_{pr} ($t'_{pr} \leq t_{pr}$). Then, the training of the model starts immediately, which gives the instances in X'_{tr} the following maximal time for training:

$$t_{tr} = t_f - t'_{pr}.$$

Such a flexible approach guarantees that the given time is used for both: training and prediction. In the case when $\vec{v_U}$ ($\vec{v_L}$) is higher than $\vec{v_M}$, the controller is aware that not all instances in X'_{pr} (X'_{tr}) can be processed by M in t_{pr} (t_{tr}). Therefore, only a specific part $X_{pr} \subseteq X'_{pr}$ ($X_{tr} \subseteq X'_{tr}$) can be processed in t_{pr} (t_{tr}), while the rest $\overline{X}_{pr} \subseteq X'_{pr}$ ($\overline{X}_{tr} \subseteq X'_{tr}$) is not provided to the model by the controller. While the controller can discard all instances from $\overline{X}_{tr}$, it is mandatory that also all instances from $\overline{X}_{pr}$ receive a value for the attribute of interest y. As there is no time left in t_{pr}, the model M cannot be applied. To guarantee a prediction, the controller assigns the average value of the attribute of interest over all processed instances $e_i \in DS_L$ ($\overline{y}$) to all instances in $\overline{X}_{pr}$.

7.4 Experimental Evaluation

This section evaluates the proposed framework on three different data streams, using three different learning algorithms. First, the data stream generation is explained. Second, two alternative frameworks are described, which are used for comparison in the following evaluation. Third, the experimental setup as well as the used learning algorithms are explained. At last, the results are shown and discussed.

7.4.1 Data Streams

Our approach is evaluated on three different data streams. To simulate the high speed data streams, 1 GB of RAM is filled with instances randomly chosen from each data source (2DimTree, Airline and Census) prior to the evaluation process. The data streams DS_L and DS_U are then created by randomly choosing instances from the main memory. Prior storage and fetching of the instances from the main memory is necessary to emulate data streams of sufficiently high speed.

7.4.2 Alternative Frameworks

There are two straightforward frameworks to handle high speed data streams with a speed higher than the instance processing speed of the model:

The *running-sushi framework (RSF)* processes only a random subset of

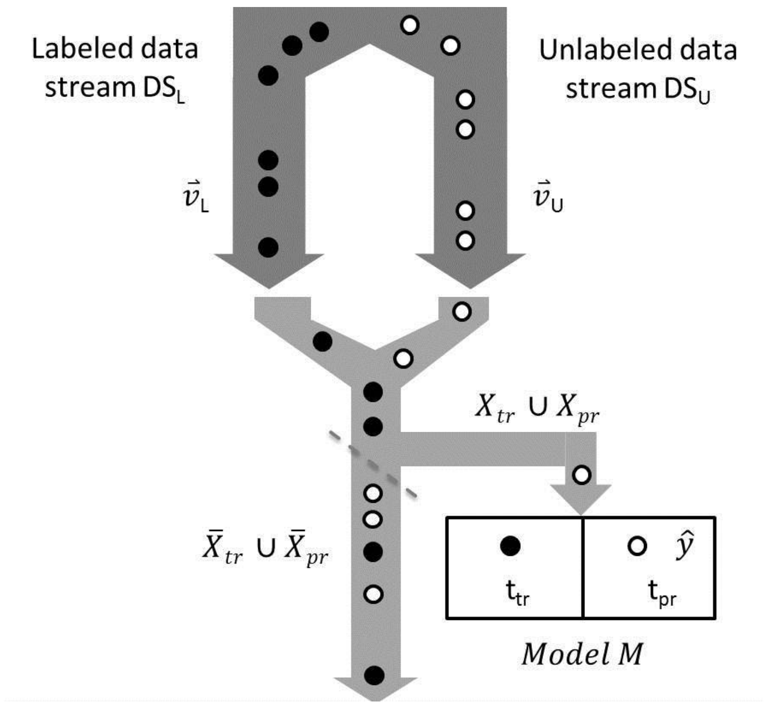

Labeled data stream DS_L
Unlabeled data stream DS_U
$\vec{v}_L$
$\vec{v}_U$
$X_{tr} \cup X_{pr}$
$\bar{X}_{tr} \cup \bar{X}_{pr}$
$\hat{y}$
t_{tr}
t_{pr}
Model M

Figure 7.2: Illustration of the running-sushi framework.

the data stream instances (sampling). It is based on the idea to fetch an instance from the data stream DS as soon as the model M is ready to process a new instance. Metaphorically, the data stream can be compared to a conveyor belt in a running sushi bar. There, you always take the next available sushi off the conveyor belt and eat it. As soon as you have finished one piece, you can take the next one. Transferring this idea to the given problem setting, the sushi belt corresponds to the data stream and each single sushi is an instance from DS_U or DS_L. The guest is the model M that processes the instances. An illustration of the framework is given by Figure 7.2. The data stream passes the model and each time the model is not processing or has finished processing an instance, the current instance in the stream is selected by the model. Labeled instances are used for training and unlabeled instances receive a prediction from the model. As long as the model is processing an instance, all arriving labeled and unlabeled instances from the stream pass.

The *queue framework (QF)* uses an external storage to process all instances by the model M. Each unlabeled instance receives a prediction, and each labeled instance is used to improve the model quality. Instances that cannot be processed by the model immediately are stored in a queue (FIFO principle) for later processing when the resources (time) are available. If the speed of the data stream exceeds the processing speed of the model,

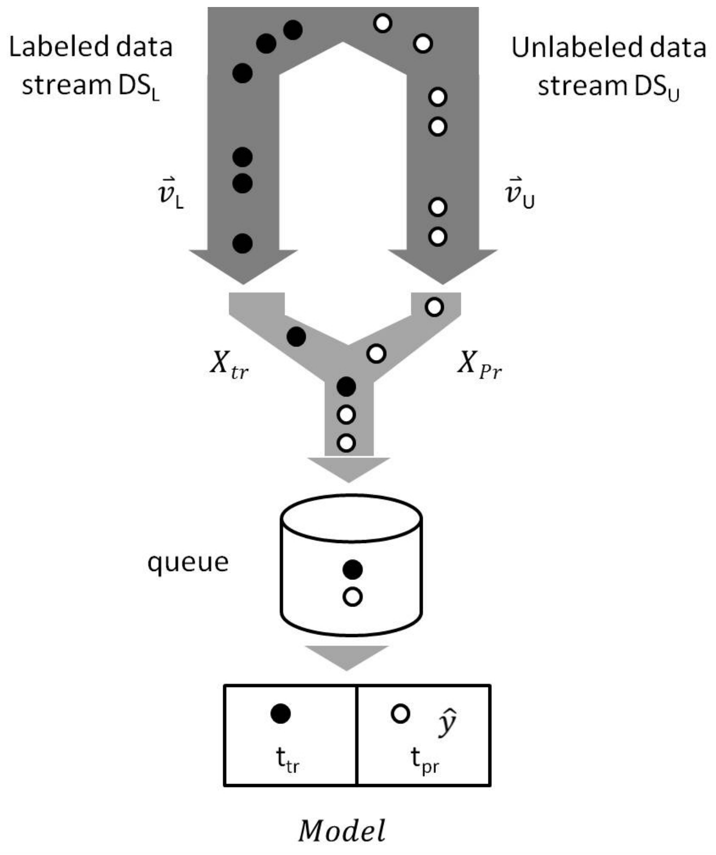


Figure 7.3: Illustration of the queue framework.

the queue extends, and if the data stream speed decreases again, the queue shrinks. The process ends when an overflow appears. An illustration of the framework is given by Figure 7.3.

7.4.3 Experimental Setup

Our framework is compared to the alternative frameworks on all three data streams. To show the flexibility and usability to integrate all kinds of incremental learning algorithms, all three approaches were used with three different incremental learning algorithms: FIMT, IMTI-RD, IMTI-RA (see Section 3.3.2 for a detailed algorithm description). The whole approach as well as the algorithms is written in JAVA. While the FIMT algorithm is a reimplementation based on the published information, the IMTI-RD and IMTI-RA algorithms are original implementations provided by the authors. All three incremental linear model trees were run with default parameters, proposed in the original publication or implementation. All runs were performed on an AMD processor with 2.6 GHz and each JAVA process was

Table 7.1: Maximal data stream processing speed (middle) and the maximal number of collected DS instances i_j until a memory exception takes place for QF (right)

Data stream	Algorithm	Max. $\vec{v}_L$	Max. $\vec{v}_U$	Max. i_j using QF (mio)
2DimTree	FIMT	434,852	2,781,893	336
	IMTI-RA	29,551	2,858,142	386
	IMTI-RD	18,612	4,370,217	402
Airline	FIMT	60,909	358,009	253
	IMTI-RA	579	1,424,470	351
	IMTI-RD	45	2,565,789	353
Census	FIMT	52,241	319,892	186
	IMTI-RA	113	801,056	271
	IMTI-RD	5	1,678,321	270

given 3800 MB of RAM. The following results are the averaged means of 5 runs using different instance orders in the streams. To motivate our approach, the maximal instance processing speed of each learning algorithm is presented first. Then, each framework is tested using each learning algorithm on all three data streams. The data stream speeds are both set to 1,700,000 instances per second ($\vec{v}_L = 1,700,000$ instances per second (ips) and $\vec{v}_U = 1,700,000$ ips). For *PAFAS*, t_f is set to 50 milliseconds. The applicability of the three different frameworks on the given high speed data streams is evaluated based on the number of processed instances, and an analysis of the response time is given. Finally, the prediction accuracy of each framework is compared.

7.4.4 Results

To test the maximal genuine instance processing speed of each learning algorithm, instances are loaded into the main memory and directly fed to the learning algorithm without any processing system in between. The number of instances that can be used for training (or processed for the prediction respectively) in a second is measured. This can be be interpreted as the maximal data stream speeds ($\vec{v}_L$ and $\vec{v}_U$) that can be processed by the algorithm (shown in Table 7.1). It can be observed that the processing speed of the labeled and unlabeled data streams are very different. Instances from the unlabeled stream (DS_U) can be processed much faster compared to instances from the labeled stream (DS_L). This can be explained by the fact that the learning algorithm is only used on the unlabeled instances X_{pr} for predictions, which is a relatively fast process. In contrast, the labeled

instances X_{tr} are used to train, i.e., to improve the model. This process can be, depending on model complexity and data dimensionality, very time-consuming. This becomes evident when comparing the processing speed of the simpler and faster FIMT algorithm to the more complex ones (IMTI-RA and IMTI-RD) over increasing stream complexity (2DimTree to airline to census). This culminates in only 5 instances per second for the IMTI-RD algorithm on the census data stream. However, in real-world applications, the learning algorithms are further embedded in a framework where the instances are fetched from the stream and delivered to them. This framework also consumes CPU time and slows down the algorithm processing speed further.
Memory problems arise for *QF* after a specific number of instances were observed. When using a data stream of 1,700,000 instances per second, which is clearly above every v_L, one can expect that *QF* will run out of memory sooner or later. This depends on the gap between $\vec{v}_M$ and $\vec{v}_U + \vec{v}_L$ and the storage size for the data stream instances. The number of instances that can be collected from the data stream until a memory exception arises is shown in Table 7.1 for our setting. These numbers suggest that using *QF* for high-speed data streams in real-world applications is not feasible as only few instances can be processed before a system crash. Contrary to *QF*, *RSF* and *PAFAS* can process instances until the framework (including the model) becomes too large for the main memory. As this time span is out of scope, the runs were stopped after fetching 4,294 billion instances (from both streams).

Processed Instances

Although an infinite number of instances could be processed by *RSF* and *PAFAS*, it is still important to process as many instances as possible from the data stream. The more instances are included in the training process, the more accurate the predictions should be. And of course in the presented setting every unlabeled instance should also receive a prediction. Thus, the first evaluation addresses the number of processed instances for each framework. *QF* is left out due to its limited usability. Table 7.2 and Table 7.3 show the number of processed instances for each framework after the forced end of each data stream.

The number of processed instances for training and testing using *RSF* is nearly equal. This is a consequence of the equal data stream speeds and for that of the equal probabilities to fetch a labeled or unlabeled instance. In contrast, *PAFAS* processes many more prediction instances than training instances. This is done by constantly collecting the instances over the time period t_f and by processing the prediction instances first. The prediction instances are thus preferred over the training instances. More instances are predicted using M, which is reflected in an improved prediction accuracy.

Table 7.2: Performance after 4.294 billion instances from the 2DimTree data stream

2DimTree			
Algorithm	Framework	#Trained	#Pred. by model
FIMT	PAFAS	885,935	110,758,335
	RSF	9,368,131	9,377,314
IMTI-RA	PAFAS	622,486	2,195,176
	RSF	6,852,348	6,859,463
IMTI-RD	PAFAS	633,274	104,145,808
	RSF	6,101,393	6,105,633
Algorithm	Framework	#Pred. with mean	#Missed
FIMT	PAFAS	2,036,241,665	0
	RSF	0	2,137,622,686
IMTI-RA	PAFAS	2,144,804,824	0
	RSF	0	2,140,140,537
IMTI-RD	PAFAS	2,042,854,192	0
	RSF	0	2,140,894,367

Furthermore, the sum of instances that are used for training and prediction by the model is larger for *PAFAS* than for *RSF*. As training time is very costly, *RSF* sacrifices many prediction instances in favor of one training instance. Therefore, the sum of processed instances is much lower than in the *PAFAS* setting.

Response Time

The next quality criterion is the response time for each framework, i.e. how long it takes for a new unlabeled instance to receive a prediction after appearing in the data stream. The response time of *RSF* is only dependent on the prediction/training time of the model, e.g., a constant response time is observed. In contrast, the *QF* response time increases with the number of instances that are stored in the queue until their prediction. In fact, a linear increase of the response time can be observed, because a linearly increasing number of instances must be processed before each new instance. This is done in constant time for each instance. Last, *PAFAS* guarantees a response time not larger than $2 * t_f$ (adjustable parameter) for each unlabeled instance. First, the instances are loaded into the controller, which lasts t_f, and then they are processed in the next time frame, which also lasts t_f. If an instance cannot receive a prediction by the model during this time, the mean target value is assigned to that instance. In both cases, the instance receives a prediction after at most $2 * t_f$.

Table 7.3: Performance after 4.294 billion instances from the airline and census data streams

Airline			
Algorithm	Framework	#Trained	#Pred. by model
FIMT	PAFAS	456,736	57,666,323
	RSF	7,404,400	7,417,562
IMTI-RA	PAFAS	382,832	75,899,407
	RSF	1,228,088	1,230,360
IMTI-RD	PAFAS	45,780	115,547
	RSF	37,837	95,054
Algorithm	Framework	#Pred. with mean	#Missed
FIMT	PAFAS	2,089,333,677	0
	RSF	0	2,139,582,438
IMTI-RA	PAFAS	2,071,100,593	0
	RSF	0	2,145,769,640
IMTI-RD	PAFAS	2,146,884,452	0
	RSF	0	2,146,904,946

Census			
Algorithm	Framework	#Trained	#Pred. by model
FIMT	PAFAS	404,284	28,300,148
	RSF	7,012,848	7,018,232
IMTI-RA	PAFAS	208,164	3,109,092
	RSF	193,335	209,721
IMTI-RD	PAFAS	5,638	164,851
	RSF	5,432	6,052
Algorithm	Framework	#Pred. with mean	#Missed
FIMT	PAFAS	2,118,699,852	0
	RSF	0	2,139,981,768
IMTI-RA	PAFAS	2,143,890,908	0
	RSF	0	2,146,790,279
IMTI-RD	PAFAS	2,146,835,149	0
	RSF	0	2,146,993,948

Prediction Accuracy

The last quality criterion adresses the prediction accuracy of the instances that have been delivered from the data stream for each framework.

However, in *QF* and *RSF* not every unlabeled instance has yet received a prediction, because instances are stuck in the queue (*QF*) or have been skipped (*RSF*). In the application domain, leaving out predictions or receiving a delayed prediction may be harmful. Therefore, a penalty is imposed for each instance that is stuck in the queue or was skipped. In both cases, the penalty is set to the average error when using $\overline{y}$ as prediction. This is equal to the non-model prediction made by *PAFAS* and corresponds to the minimal error that can be made without using a specific model. Of course, the penalty can be adapted to the severity of not predicting an instance, which can then lead to an even worse score. Figures 7.4, 7.5 and 7.6 illustrate the MAE for the predictions that have been received from the stream. *PAFAS* achieves a better MAE in 6 out of 9 cases, because more predictions are made with the model, although it was trained with fewer instances. This may nevertheless be enough for a useful prediction. The cases with an higher error often occur with IMTI-RD on more complex datasets, which could be the result of a temporarily overly strong emphasis on the prediction.

7.5 Conclusion and Future Work

This chapter proposes a framework to handle high-speed data streams consisting of labeled and unlabeled instances. It assures that each unlabeled instance receives a prediction in a bounded time interval, while the model is still constantly improved by the labeled instances. Its applicability to three learning algorithms on three data streams has been shown and its performance has been compared to two other approaches. The proposed framework focuses on a single-core implementation yet, as it is a good starting point to develop the approach towards more complex settings. Three interesting adaptations of *PAFAS* could be addressed in the future. First, the framework could be extended to multicore and distributed systems learning several models. Second, using the advantages of anytime learners, the available training and prediction time could be used more efficiently. For the training instances, sampling variants could be used to choose the most useful ones. On the prediction side, a granularity approach that dynamically decides which depth of the model should be used for the prediction could be tested. There, using sampling is not appropriate, as each instance has to receive a prediction. Third, the time frames could be partitioned into training and prediction times dynamically, as it could be tuned corresponding to the model quality. To obtain a good-quality model, as many instances from X'_{tr} should be provided, which implies that t_{tr} should be as large as possible: $t_{tr} \rightarrow t_f$. On the other hand, the user is of course interested in obtaining predictions by the model M for instances X'_{pr}, which implies that t_{pr} should be as large as possible: $t_{pr} \rightarrow t_f$. Therefore, it would be interesting to think about the integration of a model-quality dependent time-split decision, possibly using reinforcement learning.

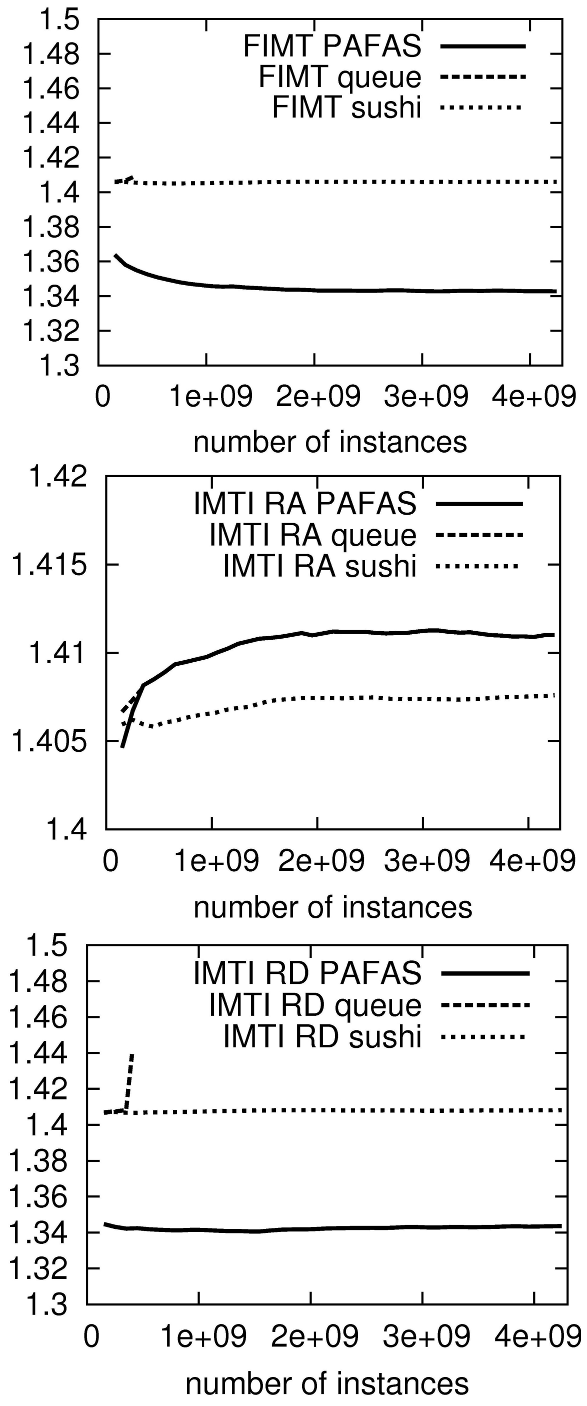


Figure 7.4: MAE development for all approaches on the 2DimTree data stream.

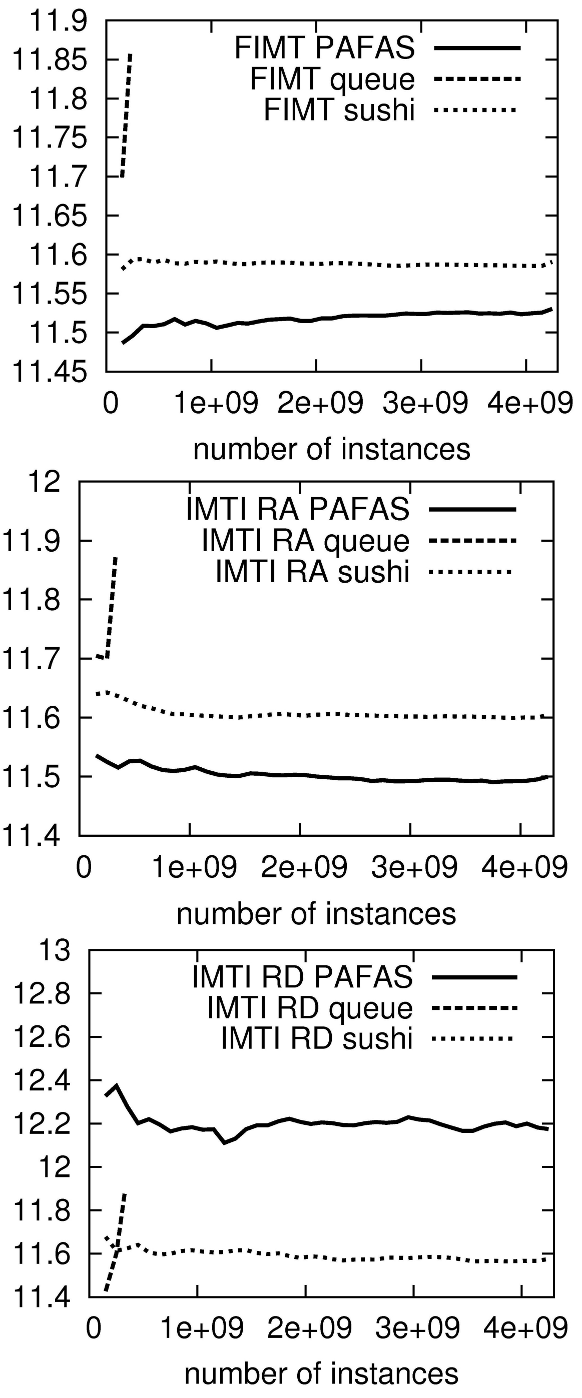


Figure 7.5: MAE development for all approaches on the Airline data stream.

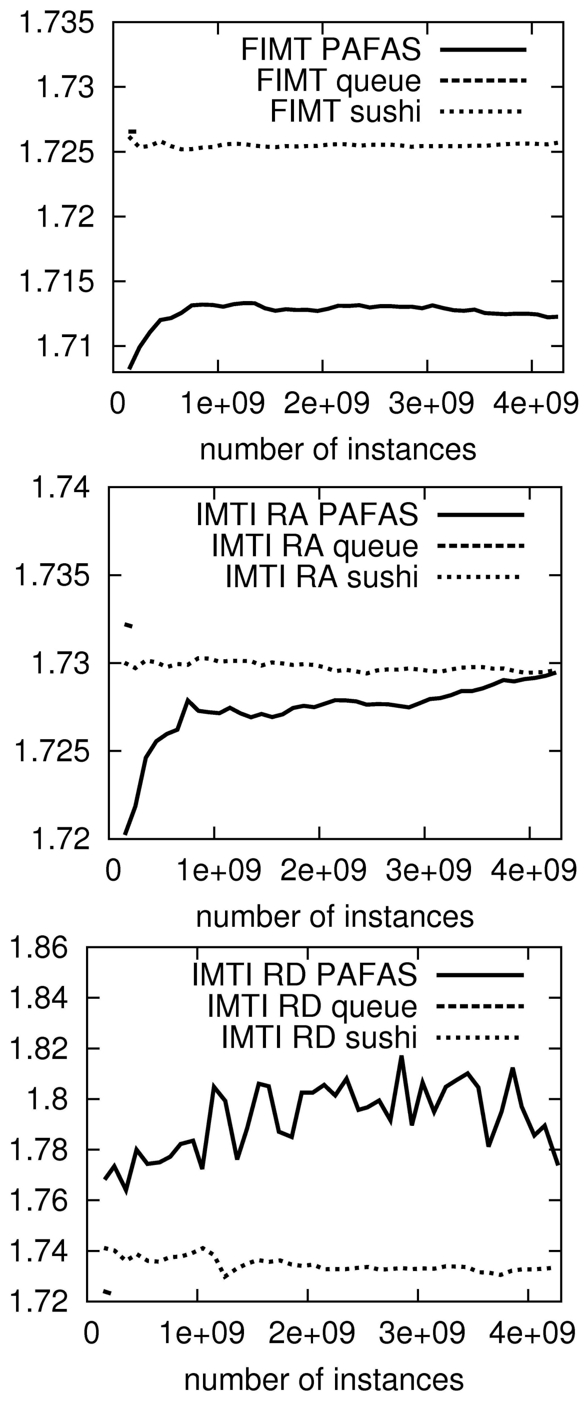


Figure 7.6: MAE development for all approaches on the Census data stream.

Chapter 8

Summary and Outlook

In the last chapter of this thesis, the main contributions are summarized and an outlook on possible future research is given.

8.1 Summary

In summary, this thesis focuses on the application of incremental linear model trees on stationary Big Data: on massive datasets, as well as on high-speed data streams. As incremental linear model trees are still a young field of research, the algorithms as well as their applications are not well studied and tested on Big Data. This thesis sheds some light on this area in a wide variety throughout the chapters:

- **Chapter 1** motivates this thesis by showing the increase of data and gives a definition of the often used term 'Big Data'. The Big Data ecosystem and especially the popular Hadoop system is explained for applications under plenty resources. The need for efficient algorithms under limited resources is further highlighted.

- **Chapter 2** gives an introduction to online and especially incremental learning. The online learning process is explained and differences to the batch learning approach are shown. Next, model evaluation approaches are introduced for the batch as well as for the online setting. The chapter closes with the description of the exemplary incremental online algorithm WINNOW.

- **Chapter 3** gives a comprehensive overview of decision trees. The regression and the classification learning tasks are explained and tree fundamentals are shown for classification and regression trees in the domain of batch and incremental learning. Decision trees are defined, their induction process, several splitting criteria, pruning and the lookahead strategy are explained in detail. Finally, an overview of decision tree algorithms is given for the classification and the regression task in the domain of batch and incremental learning. Incremental regression tree algorithms are explained in detail and their evolution from the corresponding batch algorithm is shown.

- **Chapter 4** gives an overview of the data sources used throughout this thesis.
- **Chapter 5** shows a systematic performance evaluation of incremental linear model trees on massive stationary datasets under equal conditions in three different dimensions: prediction error, running time, and memory consumption. This is done for the first time, as only insufficient analysis of ILMTs was available. Until now, related work evaluated the algorithms with too few examples to reveal their performance on massive data sources. The comparison to other ILMTs was also performed on too few examples and additionally under different conditions (e.g., algorithms in different programming languages were used). Our performance evaluation tests the algorithms within the same framework on large-scale artificial and real-world datasets, under various parameter settings. The results indicate that, first, using parameter settings that lead to simpler induction processes result in equivalent MAE in the long run and also come with the advantage of reduced running times. Additionally, on real-world datasets the algorithm with the simplest induction process, FIMT, is the fastest and most accurate algorithm. Its advantage is also still increasing with bigger datasets. Therefore, our experiments suggest that simplicity is a virtue when learning incremental linear model trees on massive datasets.
- **Chapter 6** focuses on the pruning process of incremental linear model trees. Although pruning is a standard step in the batch versions, it is a neglected research area for ILMTs. Most work was done in detecting concept drift and adapting the tree to the new concept. This includes pruning and the relearning of subtrees. Nevertheless, on massive stationary data sources, concept drift will not be available and the trees will not be pruned. This results in overly large trees where wrong split decisions are not corrected. Overly large trees are time and memory consuming and hinder the application of ILMTs on high-speed data sources. Consequently, we introduce the guarded incremental pruning approach GuIP for stationary data sources. It is an extension of incremental linear model trees with approximate lookahead in general and is exemplarily integrated into the FIMT algorithm. Results on five massive datasets show that the prediction accuracy, tree size, memory consumption and consequently, the example processing speed are influenced by the prune guard parameter ξ. By adjusting ξ, a predic-

tion accuracy gain can be achieved depending on the degree of dataset complexity. For less complex datasets, this gain can be achieved along with producing significantly smaller trees in a fraction of the time. For more complex datasets, more time is needed to achieve a better or equal accuracy. When runtime is a limiting factor, decreasing ξ can immensely speed up the learning process by a significant reduction of tree size. Moreover, for less complex datasets, a smaller ξ value may still result in a better prediction accuracy, while for more complex datasets, a marginal decrease of prediction accuracy can occur. Consequently, depending on the requirements, if processing speed or prediction accuracy is more important, the tree can be tuned accordingly. The tree size and processing speed advantage becomes even more pronounced given more and more examples without any prediction disadvantage. Additionally, the advantage over the equivalent batch algorithm has been shown on complex massive datasets. Improved prediction results are obtained in only a fraction of the time needed by the batch algorithm. Moreover, additional examples still improve the prediction accuracy of the online algorithm, which leads to a more marked advantage for even larger datasets.

- **Chapter 7** handles the problem of high-speed data streams in combination with processing-speed limited learning algorithms. Learning algorithms have an intrinsic maximal example processing speed for the learning and the prediction task. When data streams now deliver examples faster than the algorithms can process them, examples have to be skipped. Consequently, the model cannot be further improved by the skipped labeled examples, and no predictions are given for the skipped unlabeled examples from the data stream. Missing predictions are, e.g., a problem in emergency systems where prompt predictions are needed on each example. Severe consequences could have been avoided if skipped examples would have raised an alarm at early notice. For this setting, the data stream processing system PAFAS is introduced that guarantees a prediction for all unlabeled examples promptly after arrival time, while the model is still constantly improved. The system is applied to three learning algorithms on three data streams and its performance is compared to two other approaches: Temporarily storing the examples on an extra storage space (queue framework) and only randomly choosing an example when the algorithm is ready (running-sushi framework). Results show a better prediction accuracy of the algorithms in the PAFAS framework in 6 out of 9 cases. The proposed framework focuses on a single-core implementation yet, as it is a good starting point to develop the approach towards more complex settings.

8.2 Outlook

Future work can be manifold. Although this work evaluates the performance and usefulness of ILMTs on massive stationary data sources, further evaluation of the corresponding algorithms on data sources including concept drift would be interesting. The evaluation can also be transferred to other groups of online learning algorithms. As massive open data sources are rare and evaluations on this amount of data is very time consuming, most of the algorithms in this domain are not well-studied on massive data sources. Further evaluations of this kind could give important insights in the usefulness of the algorithms in practice. For the GuIP approach, the development of a more sophisticated adaptive method for choosing the optimal value for ξ should be a direction for future work. Moreover, it would be interesting to apply GuIP to the domain of evolving data streams, to evaluate its ability to handle concept drift. Additionally, one could investigate periodical pruning (pruning after processing multiple examples, instead of pruning after every single example), to further increase the processing speed. The PAFAS framework could be improved in several directions. First, the framework could be extended to multicore and distributed systems learning several models. By that, the incoming examples can be distributed between the models, raising the need for a decision which example would be beneficial for which model. This could be a valid approach if more resources are available. Another improvement could be to use the available training and prediction time more efficiently. For the training instances, sampling variants could be used to choose the most useful ones. On the prediction side, a granular approach that dynamically decides which depth of the model should be used for the prediction could be tested. There, using sampling is not appropriate, as each instance has to receive a prediction. A third improvement could be an improvement of the time frame partitioning. For a fine grained tuning of the model quality, the time frames could be partitioned dynamically into training and prediction times. It would be worthwhile to think about the integration of a model-quality dependent time-split decision, possibly using reinforcement learning.

Chapter 9

Bibliography

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The Very Large Data Base Journal*, 12(2):120–139, 2003.

[2] C. Aggarwal. A framework for diagnosing changes in evolving data streams. In *ACM SIGMOD Conference*, pages 575–586, 2003.

[3] C. Aggarwal, editor. *Data Streams - Models and Algorithms*, volume 31 of *Advances in Database Systems*. Springer, 2007.

[4] D. Alberg, M. Last, and A. Kandel. Knowledge discovery in data streams with regression tree methods. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(1):69–78, 2012.

[5] W. Alexander and S. Grimshaw. Treed regression. *Journal of Computational and Graphical Statistics*, 5(2):156–175, 1996.

[6] H. Almuallim. An efficient algorithm for optimal pruning of decision trees. *Artificial Intelligence*, 83(2):347–362, 1996.

[7] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. Technical Report 2004-20, 2004.

[8] K. Ashton. That 'internet of things' thing. *RFID Journal*, 2009.

[9] M. Aslett. NoSQL, NewSQL and Beyond: The answer to SPRAINed relational databases. http://blogs.the451group.com/opensource/2011/04/15/nosql-newsql-and-beyond-the-answer-to-sprained-relational-databases/, 4 2011.

[10] M. Basseville. Statistical methods for change detection. In H. Unbehauen, editor, *Encyclopedia of Control Systems, Robotics and Automation*. Encyclopedia of Life Support Systems (EOLSS) Publishers, 2002.

[11] M. Basseville, A. Benveniste, G. Mathis, and Q. Zhang. Monitoring the combustion set of a gas turbine. In *Proceedings of Safeprocess'94*, 1994.

[12] M. Basseville and I. V. Nikiforov. *Detection of Abrupt Changes: Theory and Application*. Prentice-Hall, Inc., 1993.

[13] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer. MOA: Massive Online Analysis. *The Journal of Machine Learning Research*, 11:1601–1604, Aug. 2010.

[14] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer. *Data Stream Mining: A Practical Approach*, 2011.

[15] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[16] M. Bohanec and I. Bratko. Trading accuracy for simplicity in decision trees. *Machine Learning*, 15(3):223–250, 1994.

[17] R. Bohn and J. Short. How much information? Report on American consumers. http://hmi.ucsd.edu/pdf/HMI_2009_ConsumerReport_Dec9_2009.pdf, 2009.

[18] R. R. Bouckaert. Choosing between two learning algorithms based on calibrated tests. In *Proceedings of the Twentieth International Conference on Machine Learning(ICML)*, pages 51–58. Morgan Kaufmann, 2003.

[19] P. S. Bradley, U. M. Fayyad, and C. Reina. Scaling clustering algorithms to large databases. In *Knowledge Discovery and Data Mining*, pages 9–15. AAAI Press, 1998.

[20] M. Bramer. Pre-pruning classification trees to reduce overfitting in noisy domains. In H. Yin, N. Allinson, R. Freeman, J. Keane, and S. Hubbard, editors, *Intelligent Data Engineering and Automated Learning (IDEAL) 2002*, volume 2412 of *Lecture Notes in Computer Science*, pages 7–12. Springer, 2002.

[21] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Chapman & Hall, New York, 1984.

[22] M. Broy. *Cyber-Physical Systems: Innovation durch softwareintensive eingebettete Systeme*. Springer, 2010.

[23] A. Cárdenas, R. Pon, and R. Cameron. Management of Streaming Body Sensor Data for Medical Information Systems. In *Proceedings of the International Conference on Mathematics and Engineering Techniques in Medicine and Biological Scienes*, pages 186–191, 2003.

[24] D. Chan, B. Krupp, and L. Wanchoo. Earth observation system. Technical report, National Aeronautics and Space Administration (NASA), 2010.

[25] P. Chaudhuri, M. Huang, W. Loh, and R. Yao. Piecewise-polynomial regression trees. *Statistica Sinica*, 4:143–167, 1994.

[26] Y. Chi, H. Wang, and P. S. Yu. Loadstar: Load shedding in data stream mining. In *In Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 1302–1305, 2005.

[27] G. C. Chow. Tests of equality between sets of coefficients in two linear regressions. *Econometrica*, 28(3):591–605, 1960.

[28] F. Chu and C. Zaniolo. Fast and light boosting for adaptive mining of data streams. In H. Dai, R. Srikant, and C. Zhang, editors, *Advances in Knowledge Discovery and Data Mining*, volume 3056 of *Lecture Notes in Computer Science*, pages 282–292. Springer, 2004.

[29] S. L. Crawford. Extensions to the CART algorithm. *International Journal of Man-Machine Studies*, 31(2):197–217, 1989.

[30] T. Dasu, S. Krishnan, S. Venkatasubramanian, and K. Yi. An information-theoretic approach to detecting changes in multi-dimensional data streams. In *Proceedings of the Symposium on the Interface of Statistics, Computing Science, and Applications*, 2006.

[31] A. P. Dawid. Statistical theory: the prequential approach. *Royal Statistical Society. Series A*, 147:278–292, 1984.

[32] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation (OSDI)*, OSDI'04, pages 10–10. Usenix Association, 2004.

[33] A. Dobra and J. Gehrke. Secret: A scalable linear regression tree algorithm. In *In Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 481–487. ACM Press, 2002.

[34] P. Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[35] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pages 71–80. ACM, 2000.

[36] P. Domingos and G. Hulten. A general method for scaling up machine learning algorithms and its application to clustering. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML)*, pages 106–113. Morgan Kaufmann, 2001.

[37] L. Douglas. The importance of big data: A definition. http://www.gartner.com/resId=2057415, 2012.

[38] B. Efron. Estimating the error rate of a prediction rule: Improvement on Cross-Validation. *Journal of the American Statistical Association*, 78(382):316–331, 1983.

[39] T. Elomaa. The biases of decision tree pruning strategies. In *Proceedings of the Third International Symposium on Advances in Intelligent Data Analysis*, IDA '99, pages 63–74. Springer, 1999.

[40] S. Esmeir and S. Markovitch. Lookahead-based algorithms for anytime induction of decision trees. In *Proceedings of the Twenty-first International Conference on Machine Learning (ICML)*, pages 257–264. Morgan Kaufmann, 2004.

[41] F. Esposito, D. Malerba, G. Semeraro, and J. Kay. A comparative analysis of methods for pruning decision trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(5):476–491, 1997.

[42] U. Fayyad and K. Irani. The attribute selection problem in decision tree generation. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 104–110. AAAI Press, 1992.

[43] A. Fern and R. Givan. Online ensemble learning: An empirical study. *Machine Learning*, 53(1-2):71–109, 2003.

[44] F. M. Fisher. Tests of equality between sets of coefficients in two linear regressions: An expository note. *Econometrica*, 38(2):361–366, 1970.

[45] M. Gaber. Data stream mining using granularity-based approach. In A. Abraham, A.-E. Hassanien, A. de Leon F. de Carvalho, and V. Snásel, editors, *Foundations of Computational Intelligence*, volume 206 of *Studies in Computational Intelligence*, pages 47–66. Springer, 2009.

[46] M. Gaber. Advances in data stream mining. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(1):79–85, 2012.

[47] J. Gama, R. Fernandes, and R. Rocha. Decision trees for mining data streams. *Intelligent Data Analalysis*, 10(1):23–45, 2006.

[48] J. Gama, P. Medas, and R. Rocha. Forest trees for on-line data. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC)*, SAC '04, pages 632–636. ACM, 2004.

[49] J. Gama, R. Rocha, and P. Medas. Accurate decision trees for mining high-speed data streams. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, KDD '03, pages 523–528. ACM, 2003.

[50] J. Gama, R. Sebastião, and P. P. Rodrigues. Issues in evaluation of stream learning algorithms. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, KDD '09, pages 329–338. ACM, 2009.

[51] J. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva. The diverse and exploding digital universe. http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf, 2008.

[52] J. Gantz and D. Reinsel. As the economy contracts, the digital universe expands. http://www.emc.com/collateral/leadership/digital-universe/2009DU_final.pdf, 2009.

[53] J. Gantz and D. Reinsel. The digital universe decade are you ready? http://www.emc.com/collateral/analyst-reports/idc-digital-universe-are-you-ready.pdf, 2010.

[54] J. Gantz and D. Reinsel. Extracting value from chaos. http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf, 2011.

[55] J. Gantz and D. Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. http://idcdocserv.com/1414, 2012.

[56] J. Gantz, D. Reinsel, C. Chute, W. Schlichting, J. McArthur, S. Minton, I. Xheneti, A. Toncheva, and A. Manfrediz. The expanding digital universe. http://www.emc.com/collateral/analyst-reports/expanding-digital-idc-white-paper.pdf, 2007.

[57] J. Gao, W. Fan, J. Han, and P. S. Yu. A general framework for mining concept-drifting data streams with skewed distributions. In *Proceedings of the 7th SIAM International Conference on Data Mining (SDM'07)*, pages 3–14, 2007.

[58] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest - a framework for fast decision tree construction of large datasets. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB)*, VLDB '98, pages 416–427. Morgan Kaufmann, 1998.

[59] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43. ACM, 2003.

[60] S. Gollapudi and D. Sivakumar. Framework and algorithms for trend analysis in massive temporal data sets. In *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management (CIKM)*, CIKM '04, pages 168–177. ACM, 2004.

[61] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques.* Morgan Kaufmann, 3rd edition, 2011.

[62] A. Hapfelmeier, C. Mertes, J. Schmidt, and S. Kramer. Towards Real-Time Machine Learning. *ECML-PKDD 2012 Workshop: Instant Interactive Data Mining*, 2012.

[63] A. Hapfelmeier, B. Pfahringer, and S. Kramer. Pruning incremental linear model trees with approximate lookahead. *IEEE Transactions on Knowledge and Data Engineering*, 26(8):2072–2076, 2014.

[64] A. Hapfelmeier, J. Schmidt, and S. Kramer. Incremental linear model trees on massive datasets: keep it simple, keep it fast. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC)*, pages 129–135. ACM, 2013.

[65] M. Hilbert and P. López. The worlds technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65, 2011.

[66] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[67] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, KDD '01, pages 97–106. ACM, 2001.

[68] E. B. Hunt, J. Marin, and P. J. Stone. *Experiments in Induction.* Academic Press, New York, 1966.

[69] E. Ikonomovska and J. Gama. Learning model trees from data streams. In J.-F. Boulicaut, M. Berthold, and T. Horváth, editors, *Proceedings*

of the 11th International Discovery Science Conference (DS), volume 5255 of *Lecture Notes in Computer Science*, pages 52–63. Springer, 2008.

[70] E. Ikonomovska, J. Gama, and S. Džeroski. Learning model trees from evolving data streams. *Data Mining and Knowledge Discovery*, 23(1):128–168, 2011.

[71] E. Ikonomovska, J. Gama, and S. Džeroski. Incremental option trees for handling gradual concept drift. *First International Workshop on Handling Concept Drift in Adaptive Information Systems: Importance, Challenges and Solutions*, Barcelona, Spain, September 24th, 2010.

[72] E. Ikonomovska, J. Gama, R. Sebastião, and D. Gjorgjevik. Regression trees from data streams with drift detection. In J. Gama, V. S. Costa, A. M. Jorge, and P. B. Brazdil, editors, *Discovery Science*, volume 5808 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2009.

[73] E. Ikonomovska, J. Gama, B. Zenko, and S. Džeroski. Speeding up hoeffding-based regression trees with options. In *Proceedings of the 28th International Conference on Machine Learning (ICML)*, ICML'11, pages 537–544. ACM, 2011.

[74] G. Inc. Google to acquire nest. http://investor.google.com/releases/2014/0113.html, 1 2014.

[75] R. Jin and G. Agrawal. Efficient decision tree construction on streaming data. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, KDD'03, pages 571–576. ACM, 2003.

[76] G. John. Robust linear discriminant trees. *Artificial Intelligence & Statistics*, pages 285–291, 1995.

[77] G. John and P. Langley. Static versus dynamic sampling for data mining. In *In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 367–370. AAAI Press, 1996.

[78] A. Karalič. Employing linear regression in regression tree leaves. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI)*, ECAI '92, pages 440–441. John Wiley, 1992.

[79] G. V. Kass. An exploratory technique for investigating large quantities of categorical data. *Applied Statistics*, 29(2):119–127, 1980.

[80] D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB)*, VLDB '04, pages 180–191. VLDB Endowment, 2004.

[81] H. Kim and W.-Y. Loh. Classification trees with unbiased multiway splits. *Journal of the American Statistical Association*, 96:589–604, 2001.

[82] J. Kirk. Will megaupload's 28 petabytes of data be deleted? http://www.computerworld.com/s/article/9225405/Will_Megaupload_39_s_28_petabytes_of_data_be_deleted_?taxonomyId=17, 2012.

[83] R. Kirkby. *Improving Hoeffding Trees.* PhD thesis, University of Waikato, 2007.

[84] J. Kivinen and H. Mannila. The power of sampling in knowledge discovery. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, PODS '94, pages 77–85. ACM, 1994.

[85] R. Klinkenberg and I. Renz. Adaptive information filtering: Learning in the presence of concept drifts. In *Workshop Notes of the ICML/AAAI-98 Workshop Learning for Text Categorization*, pages 33–40. AAAI Press, 1998.

[86] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International Joint Conference on Artificial Intelligence*, pages 1137–1143, 1995.

[87] I. Kononenko. On biases in estimating multi-valued attributes. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, IJCAI'95, pages 1034–1040. Morgan Kaufmann, 1995.

[88] R. Kothari and M. Dong. Decision trees for classification: A review and some new results. *Lecture Notes in Pattern Recognition*, 2001.

[89] A. Kouadri, A. Bensmail, A. Kheldoun, and L. Refoufi. An adaptive threshold estimation scheme for abrupt changes detection algorithm in a cement rotary kiln. *Journal of Computational and Applied Mathematics*, 259, Part B(0):835 – 842, 2014.

[90] N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, 1988.

[91] N. Littlestone. *Mistake Bounds and Logarithmic Linear-threshold Learning Algorithms.* PhD thesis, Santa Cruz, CA, USA, 1989.

[92] W.-Y. Loh. Regression trees with unbiased variable selection and interactiondetection. *Statistica Sinica*, 12:361–386, 2002.

[93] W.-Y. Loh and Y.-S. Shih. Split Selection Methods for Classification Trees. *Statistica Sinica*, pages 815–840, 1997.

[94] W.-Y. Loh and N. Vanichsetakul. Tree-structured classification via generalized discriminant analysis. *Journal of the American Statistical Association*, 83(403):715–725, 1986.

[95] P. Lyman and H. Varian. How much information? http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/printable_report.pdf, 2003.

[96] D. Malerba, F. Esposito, M. Ceci, and A. Appice. Top-down induction of model trees with regression and splitting nodes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(5):612–625, 2004.

[97] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. Hung Byers. Big data: The next frontier for innovation, competition, and productivity, 2011.

[98] M. Mehta, J. Rissanen, and R. Agrawal. Mdl-based decision tree pruning. In *International Conference on Knowledge Discovery in Databases and Data Mining (KDD)*, pages 216–221. AAAI Press, 1995.

[99] J. Mingers. An empirical comparison of pruning methods for decision tree induction. *Machine Learning*, 4(2):227–243, 1989.

[100] G. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Newsletter*, 20(3):33–35, 2006.

[101] H. Mouss, D. Mouss, N. Mouss, and L. Sefouhi. Test of page-hinkley, an approach for fault detection in an agro-alimentary production system. In *Proceedings of the Asian control conference*, pages 815–818. IEEE, 2004.

[102] K. V. S. Murthy. *On Growing Better Decision Trees from Data.* PhD thesis, The Johns Hopkins University, 1996.

[103] C. Olaru and L. Wehenkel. A complete fuzzy decision tree technique. *Fuzzy Sets and Systems*, 138(2):221–254, Sept. 2003.

[104] N. Oza and S. Russell. Experimental comparisons of online and batch versions of bagging and boosting. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, KDD'01, pages 359–364. ACM, 2001.

[105] E. Page. Continuous inspection schemes. *Biometrika*, 41(1/2):100–115, 1954.

[106] D. Potts and C. Sammut. Incremental learning of linear model trees. *Machine Learning*, 61(1-3):5–48, 2005.

[107] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[108] J. R. Quinlan. Simplifying decision trees. *International Journal of Man-Machine Studies*, 27:221–234, 1987.

[109] J. R. Quinlan. Learning with continuous classes. In *Proceedings of the 5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, 1992.

[110] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[111] R. Rastogi and K. Shim. PUBLIC: A decision tree classifier that integrates building and pruning. In A. Gupta, O. Shmueli, and J. Widom, editors, *Proceedings of the International conference of very large data bases (VLDB)*, pages 404–415. Morgan Kaufmann, 1998.

[112] R. Rastogi and K. Shim. PUBLIC: A decision tree classifier that integrates building and pruning. *Data Mining Knowledge Discovery*, 4(4):315–344, 2000.

[113] M. Raymond. How 'big' is the library of congress? http://blogs.loc.gov/loc/2009/02/how-big-is-the-library-of-congress/, 2009.

[114] S. Roberts. Control chart tests based on geometric moving averages. *Technometrics*, 42(1):97–101, 2000.

[115] F. Rosenblatt. The perceptron - a perceiving and recognizing automaton. 1957.

[116] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[117] L. Rutkowski, L. Pietruczuk, P. Duda, and M. Jaworski. Decision trees for mining data streams based on the McDiarmid's bound. *IEEE Transactions on Knowledge and Data Engineering*, 25(6):1272–1279, 2013.

[118] S. Schaal and C. G. Atkeson. Constructive incremental learning from only local information. *Neural Computation*, 10:2047–2084, 1998.

[119] J. C. Schlimmer and D. H. Fisher. A case study of incremental concept induction. In *Proceedings of the Fifth National Conference on Artificial Intellligence (AAAI)*, pages 496–501. Morgan Kaufmann, 1986.

[120] R. Schweller, A. Gupta, E. Parsons, and Y. Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*, pages 207–212. ACM, 2004.

[121] M. Severo and J. Gama. Change detection with Kalman filter and CUSUM. In L. Todorovski, N. Lavrač, and K. Jantke, editors, *Discovery Science*, volume 4265 of *Lecture Notes in Computer Science*, pages 243–254. Springer Berlin Heidelberg, 2006.

[122] C. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[123] J. Shieh and E. Keogh. Polishing the right apple: Anytime classification also benefits data streams with constant arrival times. In *Proceedings of the 2010 IEEE International Conference on Data Mining (ICDM)*, ICDM'10, pages 461–470. IEEE Computer Society, 2010.

[124] J. Short, R. Bohn, and C. Baru. How much information? Report on Enterprise Server Information. http://hmi.ucsd.edu/pdf/HMI_2010_EnterpriseReport_Jan_2011.pdf, 2011.

[125] L. Stein. The case for cloud computing in genome informatics. *Genome Biology*, 11(5):207, 2010.

[126] N. Street and Y. Kim. A streaming ensemble algorithm (sea) for large-scale classification. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, pages 377–382. ACM, 2001.

[127] M. Suknovic, B. Delibasic, M. Jovanovic, M. Vukicevic, D. Becejski-Vujaklija, and Z. Obradovic. Reusable components in decision tree induction algorithms. *Computational Statistics*, 27(1):127–148, 2012.

[128] H. Toivonen. Sampling large databases for association rules. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB)*, pages 134–145. Morgan Kaufmann, 1996.

[129] L. Torgo. Functional models for regression tree leaves, 1997.

[130] L. Torgo. Computationally efficient linear regression trees. In K. Jajuga, A. Sokolowski, and H.-H. Bock, editors, *Classification, Clustering, and Data Analysis*, Studies in Classification, Data Analysis, and Knowledge Organization, pages 409–415. Springer, 2002.

[131] P. Utgoff. ID5: An incremental ID3. In J. Laird, editor, *Proceedings of the Fifth International Conference on Machine Learning (ICML)*, pages 107–120. Morgan Kaufmann, 1988.

[132] P. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4(2):161–186, 1989.

[133] P. Utgoff, N. Berkman, and J. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29(1):5–44, 1997.

[134] P. Utgoff and C. Brodley. An incremental method for finding multivariate splits for decision trees. In *In Proceedings of the Seventh International Conference on Machine Learning (ICML)*, pages 58–65. Morgan Kaufmann, 1990.

[135] C. Vens and H. Blockeel. A simple regression based heuristic for learning model trees. *Intelligent Data Analysis*, 10(3):215–236, 2006.

[136] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11:37–57, 1985.

[137] D. Vogel, O. Asparouhov, and T. Scheffer. Scalable look-ahead linear regression trees. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, pages 757–764. ACM, 2007.

[138] F. Wang, C. Yuan, X. Xu, and P. van Beek. Supervised and semi-supervised online boosting tree for industrial machine vision application. In *Proceedings of the Fifth International Workshop on Knowledge Discovery from Sensor Data (SensorKDD)*, SensorKDD'11, pages 43–51. ACM, 2011.

[139] H. Wang, W. Fan, P. S. Yu, and J. Han. Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, KDD'03, pages 226–235. ACM, 2003.

[140] Y. Wang and I. Witten. Inducing model trees for continuous classes. In *Proceedings of the 9th European Conference on Machine Learning Poster Papers*, pages 128–137, 1997.

[141] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[142] I. Witten, E. Frank, and M. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 3rd edition, 2011.

[143] H. Yang and S. Fong. Incrementally optimized decision tree for noisy big data. In *Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, BigMine'12, pages 36–44. ACM, 2012.

[144] H. Yang and S. Fong. Stream mining dynamic data by using iovfdt. *Journal of Emerging Technologies in Web Intelligence*, 5(1), 2013.