future internet

MDPI

Article

Flow-Based Programming for Machine Learning

Tanmaya Mahapatra *,†,‡ and Syeeda Nilofer Banoo ‡

Software and Systems Engineering Research Group, Technical University of Munich, Boltzmannstraße 03, 85748 Garching, Germany; nilofer.banoo@tum.de

* Correspondence: tanmaya.mahapatra@tum.de

† Current address: Department of Computer Science and Information Systems, Birla Institute of Technology and Science, Pilani 333031, India.

‡ These authors contributed equally to this work.

Abstract: Machine Learning (ML) has gained prominence and has tremendous applications in fields like medicine, biology, geography and astrophysics, to name a few. Arguably, in such areas, it is used by domain experts, who are not necessarily skilled-programmers. Thus, it presents a steep learning curve for such domain experts in programming ML applications. To overcome this and foster widespread adoption of ML techniques, we propose to equip them with domain-specific graphical tools. Such tools, based on the principles of flow-based programming paradigm, would support the graphical composition of ML applications at a higher level of abstraction and auto-generation of target code. Accordingly, (i) we have modelled ML algorithms as composable components; (ii) described an approach to parse a flow created by connecting several such composable components and use an API-based code generation technique to generate the ML application. To demonstrate the feasibility of our conceptual approach, we have modelled the APIs of Apache Spark ML as composable components and validated it in three use-cases. The use-cases are designed to capture the ease of program specification at a higher abstraction level, easy parametrisation of ML APIs, auto-generation of the ML application and auto-validation of the generated model for better prediction accuracy.

Keywords: end-user programming; graphical flows; graphical programming tools; machine learning as a service (MLaaS); machine-learning-platform-as-a-service (ML PaaS); machine learning pipelines

Citation: Mahapatra, T.; Banoo, S.N. Flow-Based Programming for Machine Learning. *Future Internet* **2022**, *14*, 58. https://doi.org/10.3390/fi14020058

Academic Editor: Paolo Bellavista

Received: 13 November 2021
Accepted: 9 February 2022
Published: 15 February 2022

1. Introduction

Machine Learning (ML) is a scientific discipline that develops and makes use of a particular class of algorithms which are designed to solve problems without explicit programming [1]. The algorithms infer about patterns present in a dataset and learn how to solve a specific problem. This self-learning technique of computer systems has gained prominence and has vast application in the current era. The massive influx of data from the Internet and other sources creates a large bed of structured as well as unstructured datasets, where ML techniques can be leveraged to make meaningful correlations and automated decision-making. Nevertheless, ML techniques used by domain experts like traffic engineers or molecular biologists who are less-skilled programmers have to counter a steep learning curve, i.e., learn how to program and write an ML application from scratch using general-purpose, high-level languages like Java, Scala or Python. The learning curve hinders the widespread adoption of ML by researchers unless they are well-trained in programming. In response to this, we propose to equip less-skilled programmers who are domain-experts with graphical tools. In particular, we intend to support graphical specification of ML programs via a flow-based programming paradigm and support auto-generation of target code, thereby shielding the user of such tools from nuances and complexities of programming. Such graphical flow-based programming tools called mashup tools have been extensively used to simplify application development [2].

1.1. Contributions

Succinctly, the paper contributes to these aspects via:

1. We take the Java APIs of Spark ML operating on DataFrame [3], a popular ML library of Apache Spark [4–6], model them as composable components. Every component abstracts one or more underlying APIs such that they represent one unit of processing step in an ML application. The different parameters accepted by the underlying APIs are made available on the front-end while using a specific component to support easy parametrisation.
2. Development of a conceptual approach to parse an ML flow created by connecting several such components from step 1. The parsing ensures that the components are connected in an acceptable positional hierarchy such that it would generate target code which is compilable. The parsed user-flow is used to generate target ML code using principles of Model-Driven Software Development (MDSD). Model to text transformation is used, especially API based code generation techniques [7], to transform the graphical model to target code.
3. The conceptual approach is validated by designing three ML use-cases involving prediction using decision trees, anomaly detection with k-Means clustering, and collaborative filtering techniques to develop a music recommender application. The use-cases demonstrate how such flows can be created by connecting different components from step 1 at a higher level of abstraction, parameters to various components can be configured with ease, automatic parsing of the user flow to give feedback to the user if a component has been used in a wrong position in a flow and finally automatic generation of ML application without the end-user having to write any code. The user can split the initial dataset into training and testing datasets, specify a range for different model parameters for the system to iteratively generate models and test them till a model is produced with higher prediction accuracy.

1.2. Outline

The rest of the paper is structured in the following way: Section 2 summarizes the background while Section 3 discusses the related work. We give an overview of Spark and its machine learning library, i.e., Spark ML, different kinds of data transformation APIs available in Spark and our design choices to support only specific kinds of APIs in Section 4. Section 5 describes our conceptual approach to support graphical flow-based ML programming at a higher level of abstraction involving modelling of APIs as components, flow-parsing and target code generation while Section 6 describes its realization. Section 7 validates the conceptual approach in three concrete use-cases. We compare our conceptual approach with existing works in Section 8, which is shortly followed by concluding remarks in Section 9.

2. Background

2.1. Machine Learning

In classical instruction-based programming, a machine processes datasets based on predefined rules. However, in ML programming, machines are trained on large datasets to discover the inherent pattern and thereby auto-create the data processing rules. ML programming involves the usage of an input *dataset*, *features* or pieces of information in the dataset useful for problem-solving and an *ML model*. The features when passed to an ML algorithm for learning purposes outputs an ML model. ML algorithms have been broadly classified into two categories of *supervised* and *unsupervised* learning algorithms. In supervised learning, the system learns from the training dataset with labelled data to predict future events. Examples include *classification* and *regression*. Regression provides continuous prediction while classification does non-continuous prediction. Unsupervised machine learning algorithms work on unlabelled data to find unknown patterns in the data. Unsupervised learning outputs data as clusters. Clusters are a grouping of similar data from the unlabelled input. Creation of an ML model involves specific steps like data

collection and data processing, choosing a relevant ML algorithm, training the model on the training dataset, evaluating the model on testing dataset for prediction accuracy, tuning the parameters to enhance the prediction accuracy of the model and finally using the model to make predictions on new input data.

ML is often confused with Deep Learning (DL) [8]. The necessary steps involved to create a model are the same for both ML and DL. Nevertheless, there are many subtle differences between the two. First, we rely on a single algorithm in ML to predict while DL uses multiple ML algorithms in a layered network to make predictions. Second, ML typically works with structured data, while DL can work with unstructured data too. Third, in ML, manual intervention is required in the form of model parameter tuning to improve prediction accuracy while DL self-improves to minimise error and increase prediction accuracy. Fourth, DL is suited well with the availability of massive amounts of data and is a more sophisticated technique in comparison to ML.

2.2. Machine Learning Libraries

There are a plethora of ML libraries available, including TensorFlow [9,10], PyTorch [11], FlinkML [12], SparkML and scikit-learn [13], among others. TensorFlow has become one of the most prominent libraries for both ML as well as DL. It provides flexible APIs for different programming languages with support for easy creation of models by abstracting low-level details. PyTorch is another open-source ML library developed by Facebook. It is based on Torch [14], an open-source ML library used for scientific computation. This library provides several algorithms for DL applications like natural language processing and computer vision, among others via Python APIs. Similarly, Scikit-learn is an open-source ML framework based on SciPy [15], which includes lots of packages for scientific computing in Python. The framework has excellent support for traditional ML applications like classification, clustering and dimensionality reduction. Open-source computer vision (OpenCV) is a library providing ML algorithms and mainly used in the field of computer vision [16]. OpenCV is implemented in C++. However, it provides APIs in other languages like Java, Python, Haskell and many more. Apache Flink, the popular distributed stream processing platform, provides ML APIs in the form of a library called FlinkML. The application designed using these APIs will run inside the Flink execution environment. Another prominent open-source library is Weka. These are some of the most widely used libraries and listing all the available all the ML libraries is beyond the scope of this paper. We, therefore, invite interested readers to refer to [17] for more comprehensive information about ML and DL libraries.

2.3. Flow-Based Programming

Flow-Based Programming (FBP) is a programming paradigm invented by J. Paul Rodker Morrison in the late 1960s [18]. It is an approach to develop applications where program steps communicate with each other by transmitting streams of data. The data flow is unidirectional. At one point in time, only one process can work on data. Each process/component is independent of others, and hence many flows of operation can be generated with different combinations of the components. The components are responsible only for the input data that they consume. Therefore, the input/output data formats are part of the specifications of the components in FBP. The components act as software black boxes and are as loosely coupled as possible in the FBP application which provides the flexibility to add new features without affecting the existing network of FBP components.

2.4. Model-Driven Software Development

MDSD abstracts away the domain-specific implementation from the design of the software systems [7]. The levels of abstraction in a model-driven approach help to communicate the design, scope, and intent of the software system to a broader audience, which increases the quality of the system overall. The models in MDSD are the abstract representation of real-world things that need to be understood before building a system. These

models are transformed into platform-specific implementation through domain modelling languages. The MDSD can be compared to the transformation of a high-level programming language to machine code. MDSD often involves transforming the model into text, which is popularly known as *code generation*. There are different kinds of code-generation techniques like templates and filtering, templates and meta-model, code weaving and API-based code generation among others [7]. API-based code generators are the most simple and the most popular. These simply provide an API with which the elements of the target platform or language can be generated. They are dependent on the abstract syntax of the target language and are always tied to that language. To generate target code in a new language, we need new APIs working on the abstract syntax of the new target language.

3. Related Work

Literature hardly indicated any significant research work done to support graphical ML programming at a higher level of abstraction and simultaneously explaining programming concepts necessary for such. Nevertheless, there are a number of relevant works in the literature as well as products in the market which support high-level ML programming like WEKA [19], Azure Machine Learning Studio [20], KNIME [21], Orange [22], BigML [23], mljar [24], RapidMiner [25], Streamanalytix [26], Lemonade [27] and Streamsets [28] among others. Out of these, only Streamanalytix, Lemonade and Streamsets specifically deal with Spark ML. We compare our conceptual approach with these solutions in Section 9.

4. Apache Spark

4.1. APIs

A Resilient Distributed Dataset (RDD), an immutable distributed dataset [6], is the primary data abstraction in Spark. To manipulate data stored within the Spark runtime environment, Spark provides two kinds of APIs. The first is a direct coarse-grained transformation applied on RDDs directly using function handlers like a map, filter or groupby, among others. This involves writing custom low-level data transformation functions and invoking them via the handler functions. To simplify this kind of data processing, Spark introduced a layer of abstraction on top of RDD called a DataFrame [29]. A DataFrame is essentially a table or two-dimensional array-like structure with named columns. DataFrames can be built from existing RDDs, external databases or tables. RDDs can work on both structured and unstructured data while DataFrames strictly require either structured or semi-structured data. It becomes easy to process data using named columns rather than directly working with data. The second set of APIs is the DataFrame based APIs which work on DataFrames and perform data transformation. These APIs take one or more parameters for fine-tuning their operation. A further improvement over the DataFrame based APIs is the Dataset APIs of Spark which are strictly typed in comparison to the untyped APIs of DataFrames.

The RDD-based APIs operating at a low level provide fine-grained control over data transformation. However, DataFrame and DataSet APIs offer a higher level of abstraction in comparison to RDD-based APIs. The APIs are domain-specific and developers do not have to write custom data transformation functions to use them.

4.2. ML with Spark

Spark supports distributed machine learning via:

Spark MLlib

Spark MLlib has been built on top of Spark Core using the RDD abstractions and offering a wide variety of machine learning and statistical algorithms. It supports various supervised, unsupervised and recommendation algorithms. Supervised learning algorithms include decision trees, random forest, etc., while some of the unsupervised learning algorithms supported are k-means clustering, support vector machine, etc.

Spark ML

Spark ML is the successor of Spark MLlib and has been built on top of Spark SQL using the DataFrame abstraction. It offers Pipeline APIs for easy development, persistence and deployment of models. Practical machine learning scenarios involve different stages with each stage consuming data from preceding stage and producing data for the succeeding stage. Operational stages include transforming data into appropriate format required by the algorithm, converting categorical features into continuous features, etc. Each operation involves invoking declarative APIs which transform DataFrame based on user inputs [3] and produce a new DataFrame for use in the next operation. Hence, a Spark ML pipeline is a sequence of stages consisting of either a *transformer* or an *estimator*. A transformer transforms one DataFrame to another often via operations like adding or modifying columns in the input DataFrame while an estimator is used to train on the input data and output a transformer which can be an ML model.

4.3. Design Choice: API Selection

We have chosen DataFrame based APIs for supporting flow-based ML programming because the RDD based APIs operate at a low-level and necessitate the usage of custom written functions for bringing data transformations. This makes it challenging to come up with a generic method to parse a flow and ascertain if the components are connected in a way which leads to the generation of always-compilable target code. Additionally, it would demand strict type checking to be introduced at the tool-level to facilitate the creation of such flows. The other kind of API is the Dataset based APIs. These detect syntax as well as analytical errors during compile time. Nonetheless, these are too restrictive and would render the conceptual approach described in this manuscript non-generic in nature and tied to specific use-cases only. In comparison to these, the DataFrame based APIs are untyped APIs which provide columnar access to the underlying datasets. These are easy to use domain-specific APIs and detect syntax errors at compile time. The analytical errors which might creep in during usage of such APIs can be avoided by designing checks to ensure that the named columns which a specific API is trying to access are indeed received in its input. Additionally, the Spark ML library (version 2.4.4) has been updated to use DataFrame APIs. Hence, it is an ideal choice to use DataFrame APIs over RDD APIs.

5. Conceptual Approach

The conceptual approach to enable non-programmers easily specify an ML program graphically at a higher-level of abstraction makes use of principles of FBP and MDSD. It consists of several distinct steps as discussed below:

5.1. Design of Modular Components

An ML flow consists of a set of connected components. Hence, we need to design the foundational constituent or components first to realise flow-based ML programming. A component mostly does one data processing step in ML model creation and internally abstracts a specific API for delivering that functionality. However, modelling every single Spark ML API as a different component would defeat the very purpose of abstraction. With such a design, the components would have a one-to-one correspondence with the underlying APIs and programming syntax of the ML framework, making it harder for less-skilled programmers to comprehend. Hence, we introduce our first design choice by grouping several APIs as one component such that the component represents a data processing operation at a high level and is understandable to end-users. Moreover, the abstracted APIs are invoked in an ordered fashion within the component to deliver the functionality. The different parameters accepted by the APIs used inside a component are made available on the front-end as component property which the user can configure to fine-tune the operation of the component. The essential parameters of the APIs are initialised with acceptable default settings unless overridden by the user of the component.

The second design choice is making the components as loosely coupled to each other possible and the achieving tight functional cohesion between the APIs used within a single

component. This leaves space for a future extension where we can introduce new Spark ML APIs as components without interfering with the existing pool of components. To achieve this, we clearly define the input and output interface of every component and define positional hierarchy rules for each one of them. These rules help to decide whether a component can be used in a specific position in a flow or not. The flow-validation step (discussed in Section 5.2) checks this to ensure that the order in which the components are connected or rather the sequence in which the APIs are invoked would not lead to compile-time errors.

The third design choice introduces additional abstraction by hiding away parts of the ML application which are necessary for it to compile and run, written by developers when coding from scratch, nevertheless, which do not directly correspond to the data processing logic of the application. An example would be the code responsible for initialising the Spark session or closing it within which the remaining data processing APIs are invoked. Another example can be configuring the Spark session like setting the application name, configuring the running environment mode (local or cluster), specifying the driver memory size and providing the binding address among many others. We handle these aspects at the back-end to enable the end-user of such a tool to focus solely on the business logic or data processing logic of the application. The code-generator, running at the back-end (discussed in Section 5.3), is responsible for adding such crucial parts and initialising required settings with sensible defaults to the final code to make it compilable. Nevertheless, the default settings can be overridden by the user from the front-end. For example, the "Start" component (discussed in Section 5.2) is a special component used by the user to mark the start of the flow which can be configured to fine-tune and explicitly override different property values of the Spark session as discussed above.

5.2. Flow Specification and Flow-Checking

As a second step, we take the components (discussed in Section 5.1) and make them available to the end-user via a programming tool. The programming tool must have an interactive graphical user interface consisting of a palette, a drawing area called canvas, a property pane and a message pane. The palette contains all available modular components which can be composed in a flow confirming to some standard flow composition rules. The actual flow composition takes place on the canvas when the user drags a component from the palette and places it on the canvas. The component, when present on the canvas and when selected, should display all its configurable properties in the property pane. The user can override default settings and provide custom settings for the operation of a component via this pane. The flow, while being composed on the canvas, is captured, converted into a directed acyclic graph (DAG) and checked for the correct order of the connected components. The flow-checking should be done whenever there is a state change. A state change occurs whenever something changes on the canvas. For example, a new component is dragged onto the canvas from the palette or the connection between two already present components change, among other such change possibilities. Flow-checking checks the user flow for any potential irreconcilability with the compositional rules. Such a flow when passed to the back-end for code generation will generate target code which does not produce any compile-time errors. The components are *composable* in the form of a flow if they adhere to the following compositional rules:

1. A flow is a DAG consisting of a series of connected components.
2. It starts with a particular component called the "Start" component and ends with a specific component called the "Save Model" component.
3. Every component has its input and output interface adequately defined, and a component can be allowed to be used in a specific position in a flow if it is compatible with the output of its immediate predecessor and if it is permitted at that stage of processing. For example, the application of the ML algorithm is only possible after feature extraction. Hence, the component corresponding to the ML algorithm and evaluation must be connected after the feature extraction component.

When the flow is marked complete and flow-checking has completed successfully, it is passed to the back-end to start the code generation process. Figures 1 and 2 depict the flow-checking sequences for a typical ML flow leading to passing and failure of the process respectively. In the figure, it is assumed that the flow-checking begins after the flow is complete to make it easier for illustration purposes. Nonetheless, it is done whenever there is a change detected on the canvas.

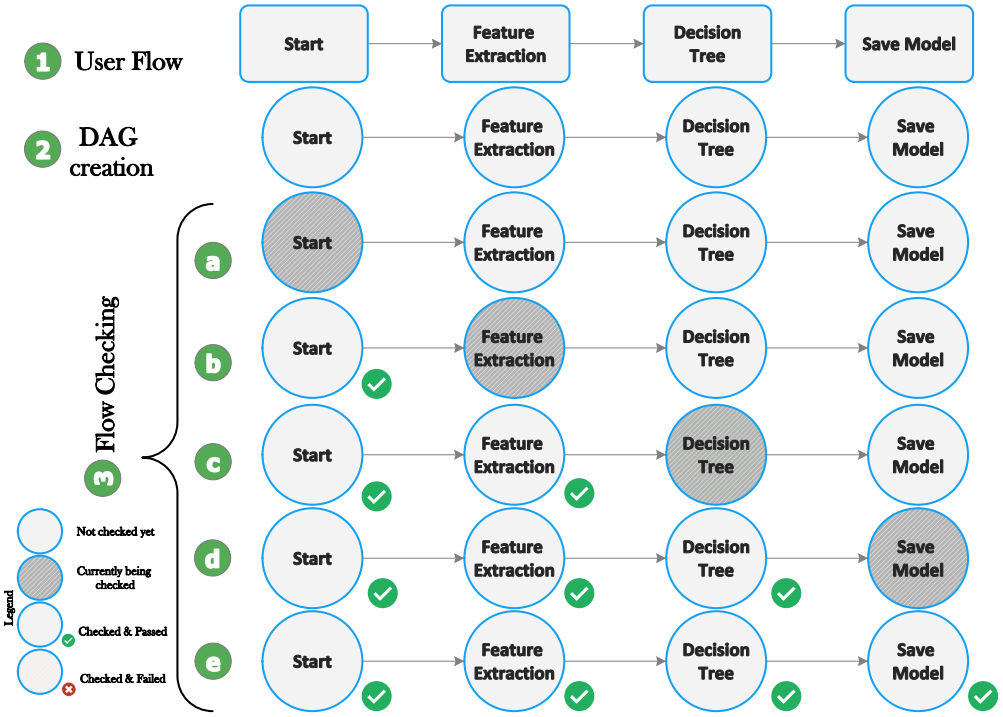

Figure 1. Flow-checking sequences for a valid ML flow.

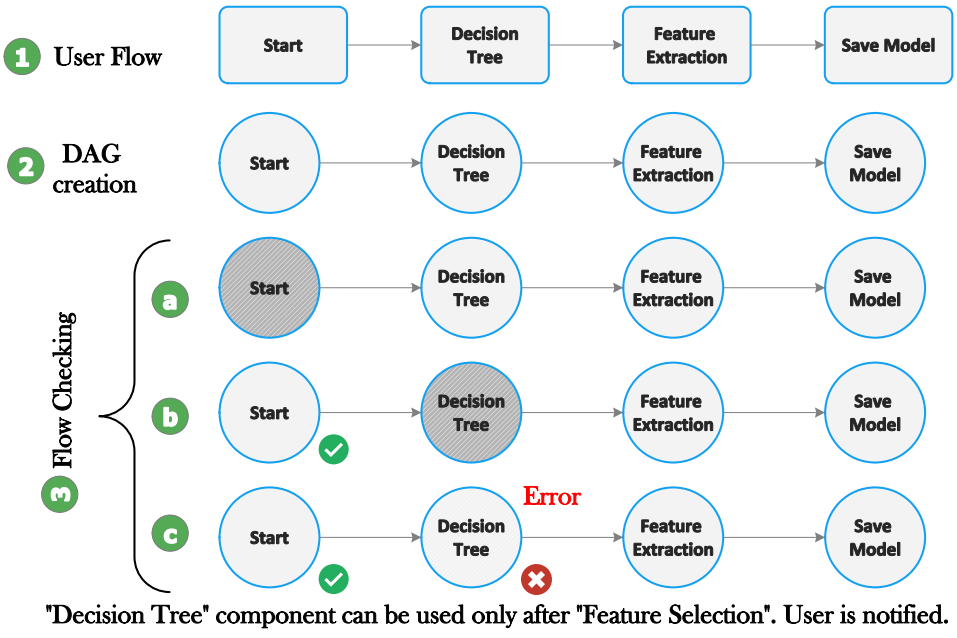

Figure 2. Flow-checking sequences for an invalid ML flow.

5.3. Model Generation

The third step deals with the parsing of the user flow and generating the target code. The back-end consists of a parser, API library and a code generator. The parser is responsible for parsing the flow received and represent it in an intermediate representation, typically in the form of a DAG. Next, it traverses the DAG to ascertain the type of component used and checks in the library for their corresponding method implementations. In MDSD, there are many code generation techniques, but we have used the API-based code generation technique because it is simple to use and serves our purpose well. The only downside is that, in an API-based code generation technique, the API can generate code only for a specific platform or language as it inherently depends on the abstract syntax of the target language to function. In our case, the API-base code generation technique is restricted to generate Java code. The method implementations for every component in the library contain statements or specific APIs to generate the particular Spark API, which is represented by the component on the front-end. The DAG or intermediate flow representation along with the information of which method to invoke from the library for each vertex is passed to the code generator. The code generator has extra APIs to generate the necessary but abstracted portion of the ML code like the start of a Spark session and inclusion of required libraries. Then, it invokes the specific method implementation from the library for each vertex, i.e., invokes the APIs contained within them to generate the target Java Spark API to be used in the final target code. When the code generator does this for all the vertices of the DAG, the target Spark ML code in Java is generated. The final code is compiled and packaged to create the runnable Spark ML driver program. The driver program is the artefact that is sent to a Spark cluster for execution.

5.4. Model Evaluation and Hyperparameter Tuning

The final step is used to test the generated model on the testing dataset to check for prediction accuracy or performance of the model. The user should have specified different model parameters to tune during flow specification. If yes, then the code generator generates different ML models, each pertaining to a unique model parameter value. All the models are tested, and the best performant ML model is saved for final usage—for example, if a user is designing an ML model using a k-means algorithm and has supplied a range of values for k as a parameter while specifying the flow, where k is the number of clusters a user wants to create from the input dataset. In this case, the code generator generates different k-means models for the range of the k values and evaluates the compute cost for each k-means model. Compute cost is one of the ways to evaluate k-means clustering using Spark ML. The system displays the k value and the corresponding model evaluation score in ascending order for the user to select or go with the best performant model. The goal of the hyper-parameter tuning in case of the k-means algorithm is to find the optimal k value for the given dataset. If the user provides only one k value, then the code generator generates only one model.

Figure 3 illustrates the conceptual approach to generate ML code after the steps of user flow specification and flow-checking have been completed. Step 1 in the figure corresponds to ideas discussed in Section 5.2, while steps 2–5 correspond to Section 5.3. Finally, step 6 corresponds to ideas discussed in Section 5.4.

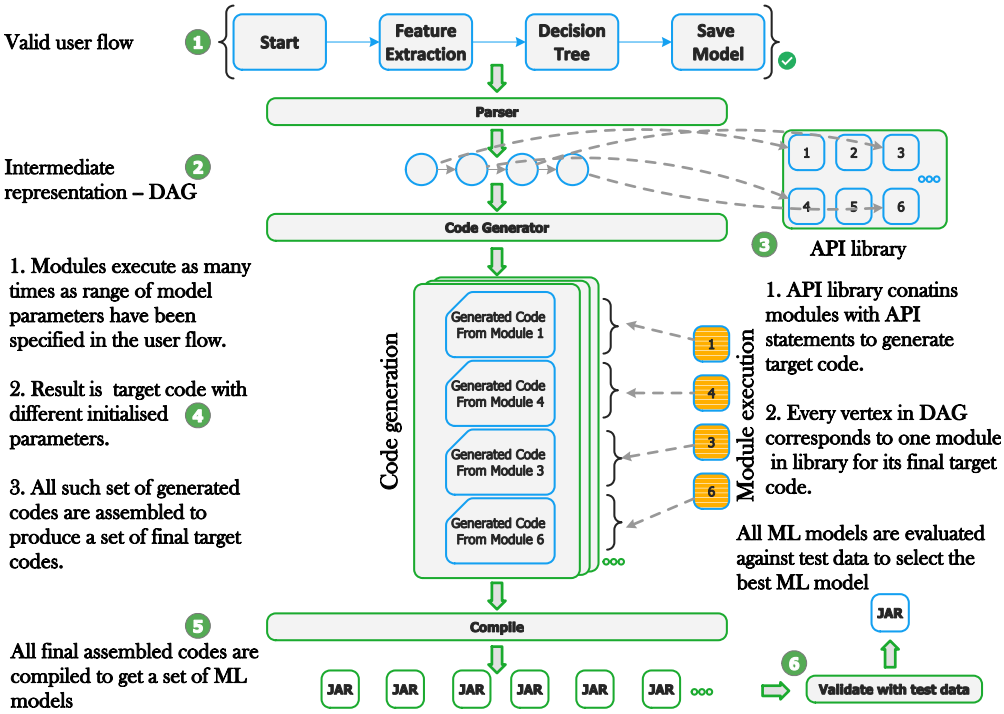

Figure 3. Generation of an ML model from a graphical user flow.

6. Realisation

In this section, we describe the realisation of the conceptual approach described in Section 5 in the form of a graphical programming tool. In particular, we describe the functioning of the tool and its architecture.

6.1. Prototype Overview

We have implemented a graphical programming tool with Spring Boot [30]. The application comes with a minimalistic interactive graphical user interface (GUI) built using Angular [31–33], a popular open-source web application framework. Figure 4 shows the GUI of the prototype with its palette containing available ML components. It also has a canvas where the components are dragged and connected in the form of a flow and, finally, a button to start the code generation process. Whenever an ML component is dragged onto the canvas, a property pane opens up allowing the user to configure its various parameters to fine-tune its operation by overriding the default values. Additionally, when something changes on the canvas, the entire flow is checked to ensure its correctness, and in case of any error, the user is provided feedback on the GUI. Figure 5 depicts these aspects of the prototype. Upon the press of the button, the flow configuration is captured in a JavaScript Object Notation (JSON) and sent to the Spring Boot back-end via REST APIs. It is converted into a Data Transfer Object (DTO) before processing. Fowler describes such a DTO as "an object that carries data between processes to reduce the number of method calls" [34]. At the back-end, the user flow is parsed, converted into an intermediate representation and passed to the code generator. The code generator auto-generates the target Spark ML application with the help of JavaPoet, an API-based Java code generator [35]. We have provided three video files demonstrating use case Section 7.1, use case Section 7.2 and use case Section 7.3, respectively, as supplementary material with this work.

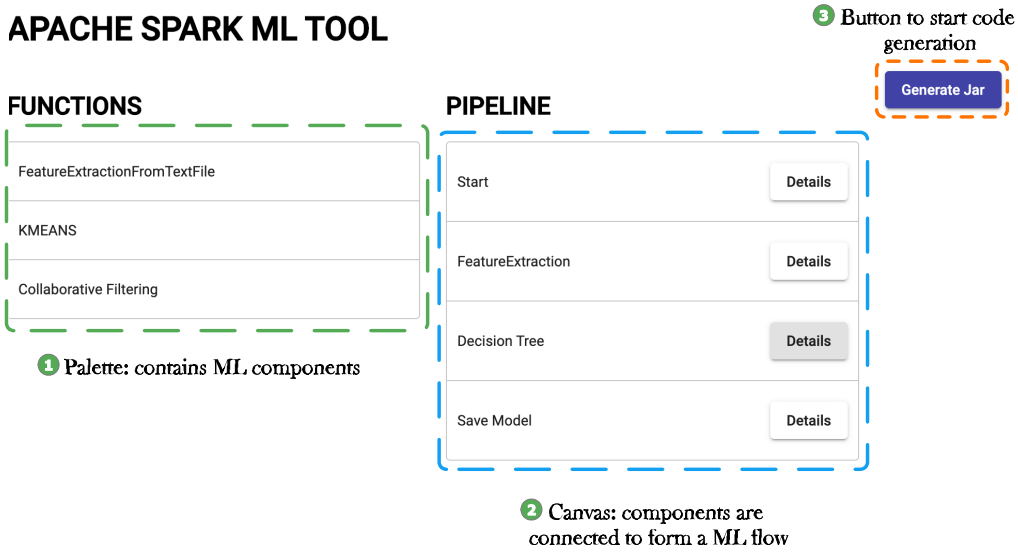

Figure 4. Interactive GUI of the prototype with palette and canvas.

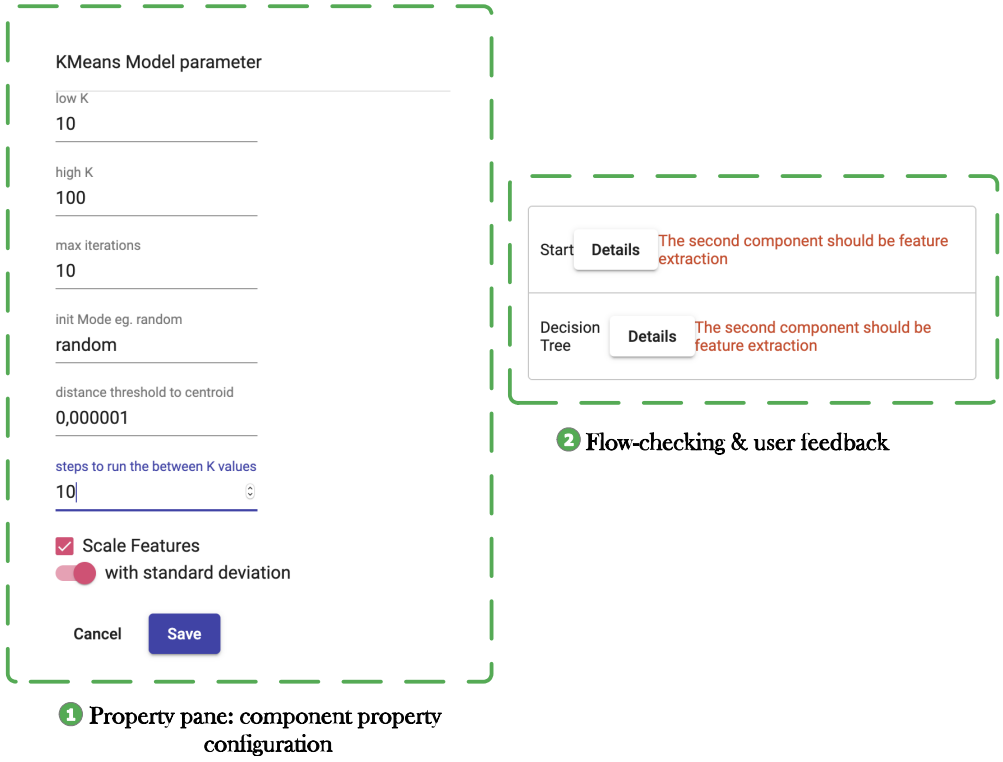

Figure 5. Component property configuration and flow-checking with user feedback.

6.2. Components

Components are the essential constituents of a flow. We have implemented around seven components relevant to our use cases out of which some of them are generic and can be used in all three of the use cases. A typical ML flow consists of four components. The first component marks the start of the flow and is responsible for the generating code related to Spark session initialisation. The second component is used to specify and pre-process the input data. It deals with dimensionality reduction, i.e., extracting a reduced set of non-redundant information form the dataset which can be fed to an ML algorithm and is sufficient to solve a given problem. The third component is a very specific training module which houses a specific ML algorithm like decision tree, k-means, among others.

The fourth component trains the model, supports hypertuning of model parameters and saves the final selected model. Table 1 lists the prototyped components and summarises their functionalities.

Table 1. Supported graphical ML components with their functionality.

SN.	Component Name	Functionality
1.	Start	Marks the start of the flow and help create a Spark session.
2.	Feature Extraction	Creates DataFrame from input data depending on the features selected.
3.	Feature Extraction From Text File	Transforms the input text data to DataFrame(s).
4.	Decision Tree	Processes the DataFrames to train a decision tree model and evaluates it.
5.	KMeans Clustering	Creates a k-means model and evaluates it. It works on given hyperparameters to find the most efficient model with minimum error in prediction.
6.	Collaborative Filtering	Creates a collaborative filtering model and evaluates it.
7.	Save Model	Marks the end of the flow and saves the ML model in a specified file path.

6.3. Working

In the back-end, the flow is parsed and represented in the form of a DAG, an intermediate representation format. It traverses all the vertices of the DAG to find out which components map to a specific module in an API library. A module in API library contains API statements written using JavaPoet, which, on execution, generates target language statements. Collection of such statements from all such modules in the library initialised with user parameters, which the code-generator assembles, results in the final target code. This code is then compiled and packaged to produce a runnable application. In this entire process, the prototype has many components which interact and accomplish certain functionalities. The system architecture of the prototype with its major components and their interactions has been summarised in the form of an Identify, Down, Aid, and Role (IDAR) graph in Figure 6.

IDAR graphs are more simple and intuitive to comprehend about the system structure, hierarchy and communication between components than a traditional Unified Modelling Language (UML) diagram [36]. In an IDAR graph, the objects at a higher level control the objects situated below. Communication is either via a downstream command message (control messages) or an upstream non-command message, which is called notice. Separate subsystems are denoted by hexagonal boxes. Similarly, a dotted line ending with an arrow depicts the data flow in the system, and an arrow with a bubble on its tail shows an indirect method call. For a comprehensive understanding of IDAR, we suggest interested readers to refer to [36,37].

The front-end has been depicted as a separate subsystem where the user creates the flow, and it is validated. When the flow is marked as complete, the front-end sends it via a REST API to the back-end controller. The controller is the main component which houses all the REST APIs for interaction with the front-end, invoking other components and coordinating the whole process of code generation. On receiving a flow from the front-end, it invokes the parser which creates a DAG out of it, saves it in the local database and traverses all the vertices to find out which specific modules in the API library must be invoked for code generation. It passes this information back to the controller, which invokes the code-generator. The code-generator invokes the necessary modules in the API library in the order of their connection in the user flow. These modules on execution produce target language statements which are then initialised with user supplied parameters. This is done for all the components the user has connected in the flow. All the generated target

statements are assembled in the same order in which the modules were invoked which is the final target code. This code is passed back to the controller. Then, the controller invokes the model generator which takes the final code compiles and packages it into a runnable Spark ML application. If the user had supplied a range of parameters for the ML model, then the code generator invokes the modules in the APIs and initialises the target statement with a different set of parameters which leads to the production of a series of final target codes. Accordingly, the model generator compiles the entire set of generated codes to produce a set of runnable Spark applications. All of the generated applications are evaluated on test data for prediction accuracy. The best performing application/ML model is selected from the whole set.

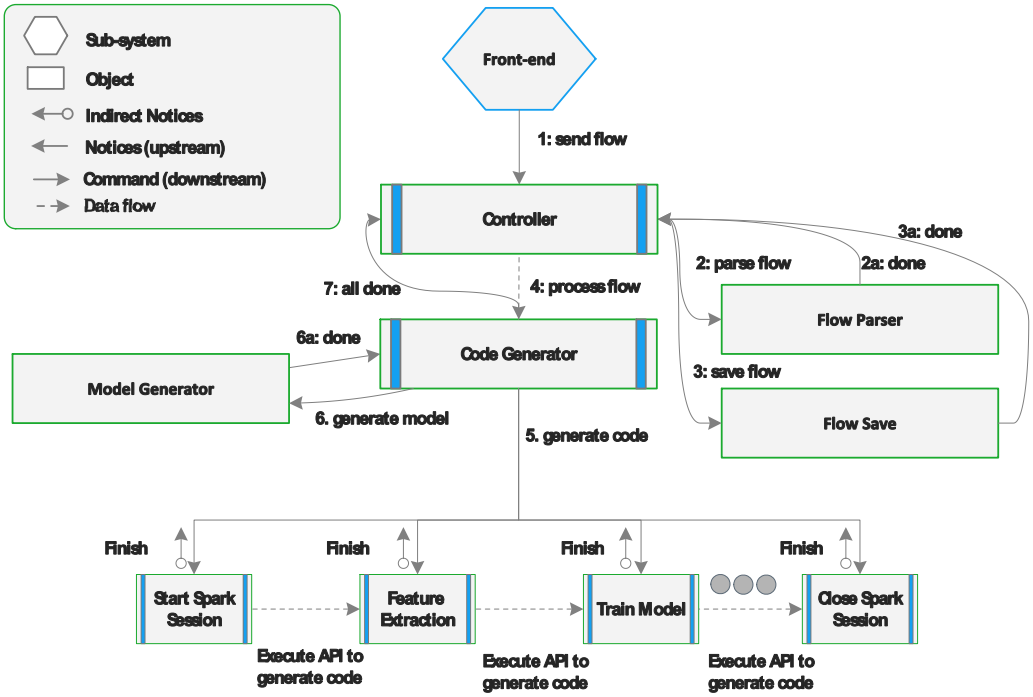

Figure 6. IDAR.

7. Running Examples

In this section, we discuss three running examples/use cases to capture the replicability of the automatic code-generation from graphical Spark flows, which is the quintessence of our conceptual approach. The running examples are based on three ML algorithms, namely—Decision Tree, k-means Clustering and Collaborative Filtering. The goal is to demonstrate that the end-user is able to create a runnable Spark program based on the ML algorithms mentioned without having to understand the detail of the Spark ML APIs or requiring any programming skills. Nevertheless, the user is expected to know the datasets on which an ML algorithm is to be applied. Additionally, the user is expected to know the label column and the features columns of the dataset.

7.1. Use Case 1: Predicting Forest Cover with Decision Trees

The first use case involved creating an ML Spark application based on the Decision tree algorithm. This supervised ML technique splits the input data depending on the model parameters to make a decision. We have used the covtype dataset in this example [38].

The dataset is available online in compressed CSV format. The dataset reports types of forest covering parcels of land in Colorado, USA. The dataset features describe the parcel of land in terms of its elevation, slope, soil type and the known forest type covers. The dataset has 54 features to describe the pieces of land and 581,012 examples. The forest covers have been categorised into seven different cover types. The categorised values range from 1 to 7.

The trained model using the 54 features of the dataset and labelled data should learn forest cover type. The data are already structured, and therefore have used it directly as input in our example.

The end-user drags various graphical components from the palette to the canvas and connects them in the form of a flow, as shown in Figure 7. The specifics of the application like the input dataset, label column and ML model parameters are taken as inputs from the user. Part 3A in Figure 7 depicts the parameters required to create a decision tree model. Internally, the parameters are specified to make calls to the Spark ML Decision Tree APIs. The flow creation specifications are sent as a JSON object to the back-end system. The decision tree model parameters comprise *impurity*, *depth of the tree* and *max bins*. The model parameters are crucial to the performance of the model. The impurity parameter minimises the probability of misclassification of the decision tree classifier. The Spark ML library provides two impurity measures for classification, namely, *entropy* and *Gini*. The impurity is set to the estimator by using the *setImpurity* function provided by *DecisionTreeClassifier* Spark API. Similarly, max bins and max depth also play a vital role in finding an optimal decision tree classifier. The end-user can tweak with these settings until the desired outcome is achieved without having to understand the internals of Spark ML APIs or updating the code manually. This process of trying out different model parameters is called hyperparameter tuning. Listing 1 lists the feature extraction function auto-generated for our use case on covtype data. Listing 2 lists the automatic generated code of the decision tree function that includes an estimator and transformer steps of an ML Spark flow.

Listing 1: Generated feature extraction code for use case 1.

```
public static Dataset<Row> featureExtraction(SparkSession spark, String filePath,
    String labelColName) {
Dataset<Row> df = spark.read()
.option("header", false)
.option("inferSchema", true).csv(filePath);

for (String c : df.columns()) {
df = df.withColumn(c, df.col(c).cast("double"));
}
df = df.withColumnRenamed(labelColName, "labelCol");
df = df.withColumn("labelCol", df.col("labelCol").minus(1));
return df;
}
```

Listing 2: Automatic generation of decision tree function.

```
public static DecisionTreeClassificationModel decisionTree(String impurity, int
    depth,
int maxBins, Dataset<Row> elbyuvroqk) {
DecisionTreeClassifier pscsjndfqw = new DecisionTreeClassifier()
.setLabelCol("labelCol")
.setFeaturesCol("features")
.setMaxDepth(depth)
.setImpurity(impurity)
.setMaxBins(maxBins);
DecisionTreeClassificationModel rwljasxoxf = pscsjndfqw.fit(elbyuvroqk);
return rwljasxoxf;
}
```

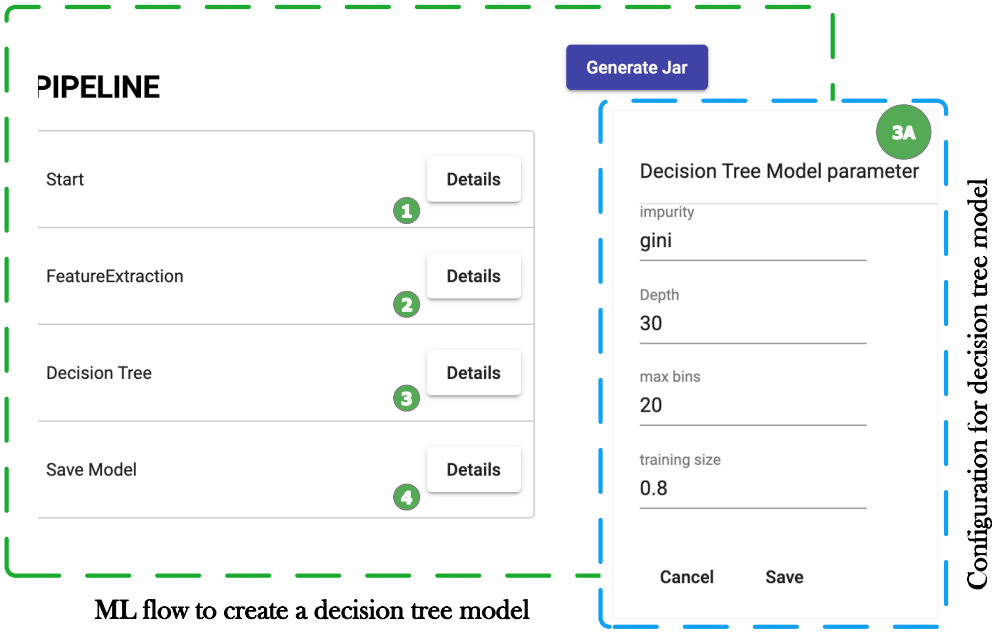

Figure 7. Graphical flow to create a decision tree model for forest cover prediction.

7.2. Use Case 2: Anomaly Detection with k-Means Clustering

The second use case deals with the creation of an ML application involving k-means. The k-means algorithm partitions the data into several clusters. Anomaly detection is often used to detect fraud, unusual behaviour or attack in the network. The unsupervised learning techniques are suitable for these kinds of problems as they can learn the pattern. We have used the data set from KDD cup 1999. The data constitutes network packet data. The data contains 38 features, including a label column. We did not need the label data for applying k-means, and therefore we removed the same in the feature extraction module. The graphical flow to create the Spark k-means application is depicted in Figure 8. The example model parameters necessary to design the model are depicted in part 3A of Figure 8. The data for the use case contained String columns. We have removed those during the feature extraction phase as the k-means algorithm cannot process them. Additionally, the String columns would cause runtime errors, as VectorAssembler only supports numeric, boolean and vector types. The code for this operation is auto-generated when the user inputs the String column names in the feature extraction stage of the flow, as depicted in part 2A of Figure 8.

The TrainModel component takes a range of K values. K is the number of clusters that the user wants to create from the input dataset. The system generates code to create k-means models for the range of the K values and stores the compute the cost for each k-means model evaluation in an array in ascending order. Compute cost is one of the ways to evaluate k-means clustering using Spark ML. If the end-user provides only one K value, the system generates only one code to generate one model. The output of the k-Means Spark application for automatic hyperparameter tuning is shown in part 5 of Figure 8. The figure indicates different K values and their corresponding model evaluation score. The goal of the hyperparameter tuning in the case of the k-means algorithm is to find the optimal K value for the given dataset. Listing 3 shows code generated by the back-end system for the graphical flow composed on the front-end.

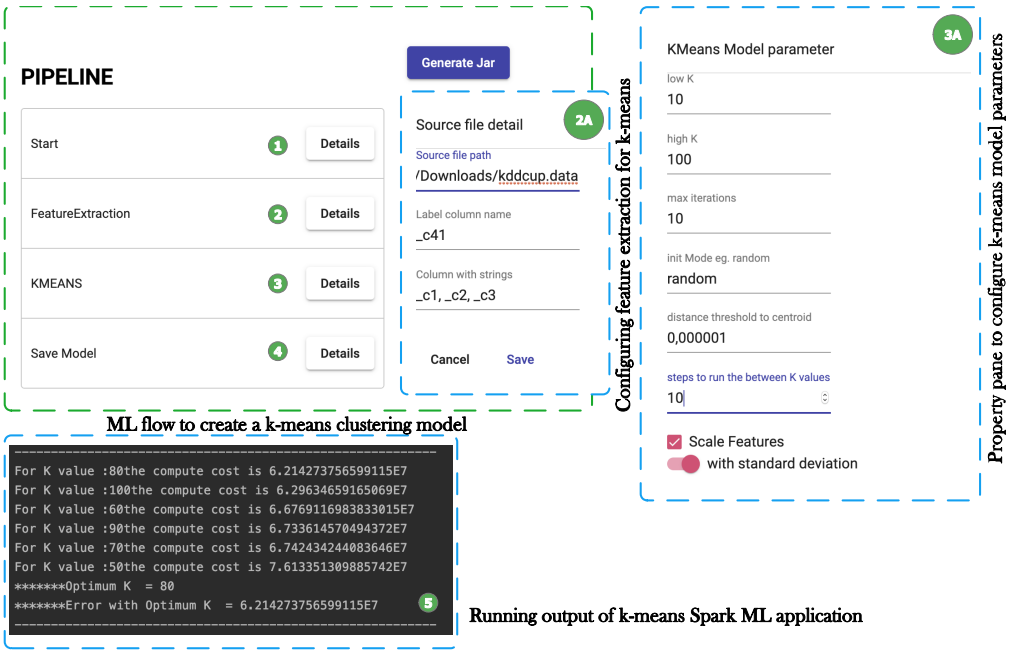

Figure 8. Graphical flow to create a k-means clustering model for anomaly detection.

Listing 3: Function generated for applying KMeans algorithm.

```
public static KMeansModel kMeansClustering(String initMode, int lowK, int highK,
    int maxIter,
double distanceThreshold, int step, Dataset<Row> oiosnsutcx) {
Map<Integer, Double> degklrfhyb = new LinkedHashMap<Integer, Double>();
Map<Integer, KMeansModel> mnqoxuuyhr = new LinkedHashMap<Integer, KMeansModel>();
for(int iter = lowK; iter <= highK; iter +=step) {
KMeans pjtnfmyssi = new KMeans().setFeaturesCol("features")
.setK(iter)
.setInitMode(initMode)
.setMaxIter(maxIter)
.setTol(distanceThreshold)
.setSeed(new Random().nextLong());
KMeansModel okvhgrwqrk = pjtnfmyssi.fit(oiosnsutcx);
// Evaluate clustering.
Double ujyyyccvqv = okvhgrwqrk.computeCost(oiosnsutcx);
degklrfhyb.put(iter, ujyyyccvqv);
mnqoxuuyhr.put(iter, okvhgrwqrk);
System.out.println("*******Sum of Squared Errors = "+ ujyyyccvqv);
}
Map<Integer, Double> vqtkeniici = degklrfhyb.entrySet()
.stream()
.sorted(comparingByValue())
.collect(toMap(Map.Entry::getKey, Map.Entry::getValue, (e1, e2) -> e2,
LinkedHashMap::new));
Integer hwnnwdbhqd = vqtkeniici.entrySet().stream().findFirst().get().getKey();
KMeansModel okvhgrwqrk = mnqoxuuyhr.get(hwnnwdbhqd);
System.out.println("*******Optimum K = "+ hwnnwdbhqd);
System.out.println("*******Error with Optimum K = "+ degklrfhyb.get(hwnnwdbhqd));
return okvhgrwqrk;
}
```

7.3. Use Case 3: Music Recommender Application

The third use case demonstrates the application of Spark Collaborative filtering APIs. The use case checks the code generation of loading data from text files and creating a custom "Rating" class from the input text file. The aim is to ensure that the generated application should load the input text file using the "Rating" class, train and evaluate the recommender model according to the end-user preferences. For this use case, we have used data published by Audioscrobbler, the first music recommendation system for last.fm. The

input data include three text files, user_artist_data.txt, artist_data.txt and artist_alias.txt. The primary data file is user_artist_data.txt, which contains user id, artist id and play count. The artist_data.txt file includes the names of each artist mapped to artist ID. The artist_alias.text contains the map artist ID that is unknown misspellings. The data contain implicit feedback data; it does not contain any direct rating or feedback data from users.

Collaborative filtering (CF) is a model-based recommendation algorithm. The CF algorithm finds the hidden factors or latent factors about the user's preferences from the user's history data. The current data set fits for collaborative filtering application as we do not have any other information about users or artists. These type of data are sparse. The missing entries in the user-artist association matrix are learnt using the alternating least square (ALS) algorithm. At the moment, the Spark ML library supports only model-based collaborative filtering. We have used the Spark ML ALS estimator to train the recommendation system. The Spark ML collaborative filtering requires the user to develop a Rating Java class for parsing the main input data file while loading the raw data into a DataFrame. The Rating class is responsible for casting the raw data to respective types, and, in our case, we also need the implementation to map the misspelt artist IDs to correct IDs. spark.ml package also provides APIs to set the cold start strategy for NaN (Not a Number) entries in the user data to mitigate the cold start problem. It is quite normal to have missing entries in such data; for example, maybe a user never rated a song, and a model can not learn about the user in the training phase. The dataset used for the evaluation may have entries for users that are missing in the training dataset. These problems are defined as a cold start problem in recommender system design.

The CM ML flow features extraction technique is a bit different from the previous two use cases. The Spark ML implementation requires the system to generate a Rating class file in Spark version 2.4.4. The Rating file adds better control to the model designing. Listing 4 shows the Rating class auto-generated by the back-end system. Figure 9 depicts the assembling of graphical components required for CF application generation. The CF Spark program generation required the FeatureExtractionFromTextFile component.

Listing 4: Rating class for parsing CF input data.

```
@Getter
@Setter
public static class Rating implements Serializable {
private Integer userId;

private Integer playcount;

private Integer artistId;

public Rating(Integer artistId, Integer playcount, Integer userId) {
this.artistId = artistId;
this.playcount = playcount;
this.userId = userId;
}

public static Rating parseRating(String str) {
String[] fields = str.split(" ");
if (fields.length != 3) {
throw new IllegalArgumentException("Each line must contain 3 fields");
};
Integer artistId = Integer.parseInt(fields[0]);
Integer playcount = Integer.parseInt(fields[1]);
Integer userId = Integer.parseInt(fields[2]);
Integer finalArtistData = CollaborativeFiltering.artistAliasMap.getOrDefault(
    artistId, artistId);
return new Rating(artistId,playcount,userId);
}
}
```

The user can create an application with auto-hyperparameter tuning through the graphical interface shown in part 3A of Figure 9. All the available Spark ALS API parameters' settings have been considered to design the collaborative filtering module in our system. In the given dataset, the preference of the user is inferred from the given data. Hence, the implicit preference box should be checked, or in case of sending direct JSON data, the field should contain the true value. Part 3A of Figure 9 depicts how to set the model parameters for CF application. The model parameters are crucial in making a sound recommender system. The user interface allows for setting different parameters for training the model.

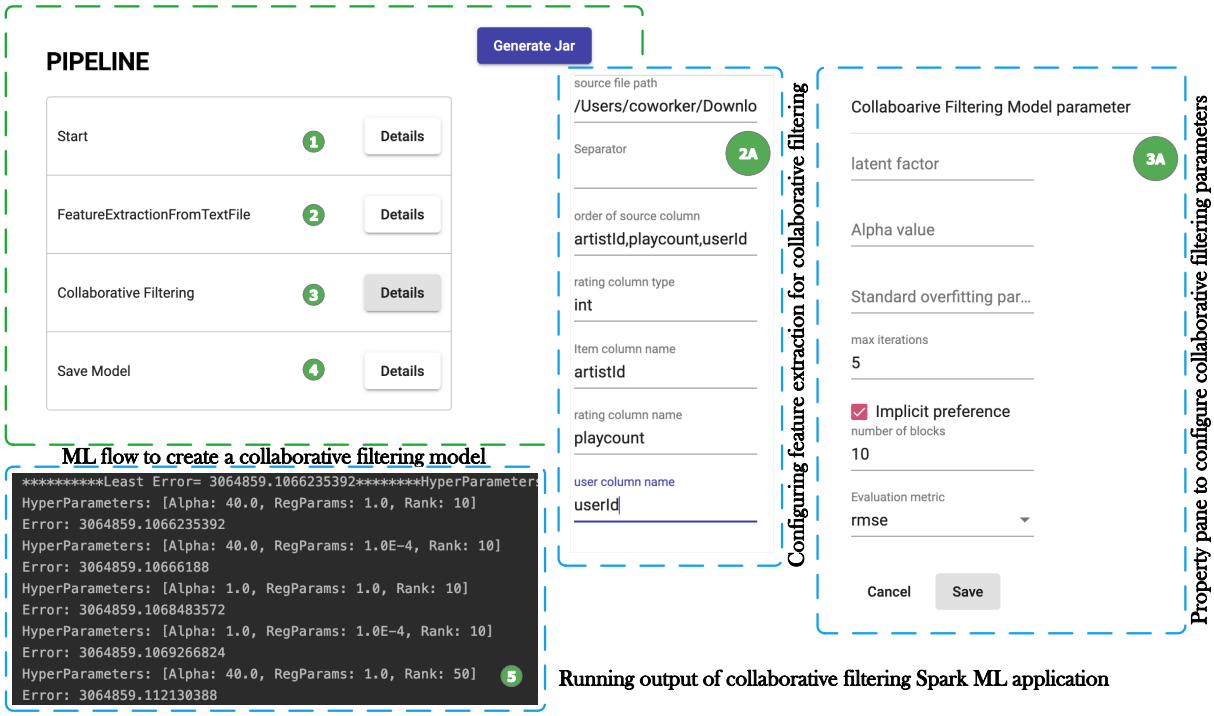

Figure 9. Graphical flow to create a collaborative filtering model for music recommendation.

The code generation for training the model using ALS is done depending on the input parameters from the enduser. Table 2 shows the available option for the end-user, and Listing 5 shows how the parameters are set for the ALS algorithm in Spark internally.

Table 2. Supported graphical ML components with their functionality.

SN.	Parameter Name	Purpose
1.	numBlocks	parallelizing the computations by partitioning the data. default value is 10.
2.	rank	number of latent factors to be used in the model. default value is 10
3.	maxIter	maximumn number of iteration to run the train model. default value is 10
4.	regParam	regularization parameter.default value is 1.0
5.	implicitPrefs	to specify if the data contain implicit or explicit feedback. default value is false, which means explicit feedback
6.	alpha	Only applicable when implicitPrefs is set to true. default value is 1.0
7.	userCol	setting of input data user column name in the ALS algorithm
8.	itemCol	setting of input data item col name in the ALS algorithm.in our use case, its artistId
9.	ratingCol	setting of input data rating column name in the ALS algorithm

Listing 5: Example of ALS model creation and evaluation.

```
//rest of the code
for( Integer rank: ranks){
for( Double regParam: regParams) {
for( Double alpha: alphas) {
ALS als = new ALS()
.setMaxIter(10)
.setAlpha(alpha)
.setRegParam(regParam)
.setImplicitPrefs(true)
.setRank(rank)
.setUserCol("userId")
.setItemCol("artistId")
.setRatingCol("count");
ALSModel alsModel = als.fit(training);
alsModel.setColdStartStrategy("drop");
Dataset<Row> predictions = alsModel.transform(test);
RegressionEvaluator evaluator = new RegressionEvaluator()
.setMetricName("rmse") //rmse, mse, mae, r2
.setLabelCol("count")
.setPredictionCol("prediction");
Double rmse = evaluator.evaluate(predictions);
System.out.println("Hyper Params = (" + rank + " " + alpha + " " + regParam + ")")
    ;
System.out.println("Root-mean-square error = " + rmse);
}
}
}
```

Spark 2.4.4 has four variants of metrics for ALS model evaluation, namely rmse (Root Mean Square Error), mse (Mean Square Error), mae (Mean Absolute Error) and r2 (RSquared). The RegressionEvaluator uses one of these metrics to evaluate the model. The setting of the metric to this API is shown in Listing 5, and the output of the model evaluation is shown in part 5 of Figure 9. The Spark application outputs the least error and the corresponding model parameters used for ALS model creation. The output also shows the error calculated for another combination of parameters sent as input by the end-user. The end-user was able to create a runnable Spark application using a collaborative filtering algorithm with the given dataset. The user could customise the recommender system by providing different parameters to the flow components, which did not change or add any new source code to the existing system. The following observations summarise the essence of the running examples:

1. The modular approach of the system development helps add new ML functionality without affecting the existing behaviour of the system. The end-user can design applications from the list of the components provided through REST API interfaces. The components are independent of each other, and interaction between the components happen only through the input/output data to and from the components that help the end-user design Spark applications depending on the problem statement.
2. The flow-based programming approach hides the underlying Spark implementation from the end-users. The end-users do not have to learn Apache Spark ML library or functionality of Spark Data abstractions to implement an ML application using flow-based ML programming. The end-users can customise the Spark application by providing the specifications through the flow components. The end-users should only have a better understanding of the input dataset used for training a model. The flow-based programming makes it easier for users to customise their ML applications by providing specification through the graphical components.

8. Discussion

In this section, we compare the conceptual approach of our flow-based programming for ML with the existing solutions as introduced in Section 3. The comparison criteria include:

1. *Graphical interface:* The graphical interfaces should generally be intuitive and easy to use for the end-users to navigate through a software application. The graphical programming interface with flow-based programming paradigm and options to customise the automatic code generation of ML application makes an ideal choice for users with less knowledge of data science or ML algorithms. As discussed in Section 3, we have seen that almost all existing tools have a flow-based graphical interface implementation for creating an ML model, except Rapidminer. However, Rapidminer provides a graphical wizard to initialise the input parameters. In our approach, we have a list of graphical components representing the steps of ML model creation. The drag and drop feature with feedback on the incorrect assembly of the components guides the user for submitting a logical flow to the back-end system. The graphical interface implemented supports customisation of the components and abstracts the underlying technologies used for automatic Spark application generation for the user.
2. *Target Frameworks:* The second criterion is the target frameworks used by the graphical tools that provide a high-level abstraction. The Deep learning studio offers an interface to implement only deep learning models. Both Microsoft Azure and Rapidminer visual platforms support an end-to-end automated ML application generation, hiding the underlying technology used from the user. However, the Microsoft Azure HDInsight service allows PySpark code snippet to be used through the Jupyter notebook interface of the tool to run the code on a Spark cluster. Lemonade and StreamSets support high-level abstraction of Spark ML to build ML models. Lemonade also uses the Keras platform to generate models for deep learning applications. On the contrary, the Streamanalytix tool uses multiple frameworks to create an ML model, such as Spark MLlib, Spark ML, PMML, TensorFlow, and H_2O. In as our target framework.
3. *Code generation:* The code generation capability is another exciting feature to compare whether the tool can create a native program in the target framework from the graphical flow created by the end-user or not. StreamSets and Microsoft Azure require customised code snippets from end-users to train the ML model. They do not generate any code, and the models are loaded directly in their environment. Lemonade and Streamanalytix generate a native Apache Spark program, while Deep learning studio generates code in Keras from the graphical flow generated by the user. The user can edit the generated code and re-run the application. Our conceptual approach also generates Java source code for the Apache Spark program from the graphical flow created by the end-user.
4. *Code snippet input from user:* It is desirable to have a graphical tool that can support both experts and non-programmers to create ML applications without having to understand the underlying technology. The fourth criterion is to compare if the tool requires code snippets from the user for ML application creation. Mainly, the code snippet is required for some part of the application generation or the customisation of the program. For example, the StreamSets tool provides an extension to add ML feature by writing customised code in Scala or Python to the pipeline for generating the program. Tools like Rapidminer, Lemonade, Deep learning studio, and Streamanalytix do not require any input code snippet from the user to create the ML application program. While the Microsoft Azure auto ML feature does not require any code-snippet from the user, it explicitly asks for a code snippet to create models to run in the Spark environment. The conceptual approach described in this manuscript does not require the user to write any code for Spark ML application generation.
5. *Data pre-processing:* As we already know, the performance of the ML model hugely depends on the quality of the input data. The collected data are usually not structured, requiring a bit of processing before applying the ML algorithms to the data. The manual preprocessing of these data is time-consuming and prone to errors. Having a visual pre-processing feature to the ML tool saves much time for the users. All the tools except Deep learning studio and Lemonade have a data pre-processing step that

supports data cleansing through the graphical interface. Our conceptual approach also helps essential cleansing and feature extraction from the input data.

6. *Ensemble Learning:* The sixth criterion is to compare whether the tools provide the ensemble learning method. Ensemble methods is a machine learning technique that combines several base models to produce one optimal predictive model. The ultimate goal of the machine learning technique is to find the optimum model that best predicts the desired outcome—tools like Streamanalytix, Rapidminer and Microsoft Azure auto ML support the ensemble learning method. This is a limitation in our current approach as it cannot automatically combine several base models to solve a specific use case.

Table 3 summarises the comparison of our conceptual approach with the existing solutions for the criteria discussed above.

8.1. Comments about Previous Attempts

Previously, we had attempted to support programming of Spark applications via graphical flow-based programming paradigm [39–42]. The previous work culminated in the doctoral dissertation of the last author. This work is an extension of the previous work. The main difference lies in the code-generation technique. Previously, we had used the API-based code generation technique to generate only the basic skeleton of the Spark application. A library called 'SparFlo' [42] was developed, which contained a generic method implementation of various Spark APIs. Codeweaving was used to invoke these generic method implementations inside the basic skeleton of the target Spark program. This ensured that the SparFlo library, when supported by any graphical programming tool, would easily support Spark programming. Nevertheless, any changes or updates in Spark libraries would cause the release of a new version of the SparFlo library containing the latest generic method implementations of the Spark APIs. Hence, in this attempt, we have relied only on the API-based code generation technique, which eliminates our conceptual approach to develop a pre-packaged implementation of all Spark APIs. It also decouples from a specific Spark version as now we can independently parse a Spark version to generate relevant target source code.

Table 3. Comparison of existing solutions with our approach to automate ML application creation.

Tools	Graphical Interface	Target Framework	Code-Snippet Not Required as Input	Code Generation for ML Program	Include Data Pre-Processing	Ensemble Method Supported
Deep Learning Studio	Flow-based GUI	Keras	✓	✗	✗	✗
Microsoft Azure ML	Flow-based GUI	Spark ML (Python, R)	✓ (✗ for auto ML)	✗	✓	✓ (auto ML), ✗ (for Saprk application)
Streamanalytix	Graphical wizard-based	Spark ML, H2O, PMML	✓	✗	✓	✓
StreamSets	Flow-based GUI	Spark ML (Python & Scala)	✓	✗	✓	✗
Rapidminer	Graphical wizard-based	Unknown	✓	✗	✓	✓
Lemonade	Flow-based GUI	Spark ML (PySpark)	✗	✓	✗	✗
Our Solution	Flow-based GUI	Spark ML (Java)	✓	✓	✓	✗

8.2. Advantages and Limitations

In comparison to the tools widely used in practice, the approach described here offers two benefits. First, our conceptual approach uses API-based code generation techniques which make the code generator (i) generic, i.e., not tightly coupled to the target framework specifics and (ii) scalable, i.e., it can be easily configured to generate target code for a

different framework with minimal modification. The code generation process of the state-of-the-art tools is either non-existent in literature or not disclosed as in the case of Streamanalytix, RapidMiner or Azure ML. This prevents carrying out performance analysis of generated code from such graphical flow-based programming tools. Additionally, the concept of supporting graphical end-user programming, especially in the area of Big Data analytics and ML, is on the rise. One such upcoming research project is Gaia-X, a decentralized federated and secure data infrastructure where such tools can be supported to perform easy data analytics by data consumers. The absence of scalable conceptual approach for flow-based programing in this domain forces either to stick to the proprietary ones or reinvent the wheel. Second, using the code generator, we generate the target code which can be examined before execution. This has two-fold advantages. Experienced developers too can use such tools to quickly prototype their solution, generate the code and refine the generated code as per their requirements. Additionally, with the generated code available, we can subject it to performance analysis and manual code inspection to detect smells or presence of anti-patterns. Third, our approach does the data pre-processing for the end-user and does not require inputting any code snippets during flow creation. This makes the process easy for inexperienced ML users to develop their ML applications. Our approach has limitations too. We do not support ensemble learning in our approach, which is supported by most other tools. We believe this can be easily integrated into our approach. We have not carried out performance evaluation of generated code with other tools because most of the tools do not generate target code. Additionally, in cloud based solutions, the execution environment is not configurable to a large extent. In order to carry out performance evaluation, we need target codes generated by different tools for a specific ML use case and also compare it in a uniform execution environment to arrive at reasonable conclusions.

9. Conclusions

The field of data science is challenging in many ways. First, the datasets are usually messy, and most of the time, the data scientists go into pre-processing the data and selecting features from the data. Second, ML algorithms apply multiple iterations to the dataset to train the model. The process of preparing a model is computationally expensive and time-consuming. Third, the real-world application of the generated model to the new data, such as fraud detection, recommends that models become part of the production service in real-time. The data scientists engage most of their time in understanding and analysing the data. They would want to try different ML applications and tweak the models to achieve the sought accuracy in data analytics. The modelling of such ML applications adds another level of difficulty. There should be a way to reuse the models while experimenting with the existing designed systems. Apache Spark framework combines both distributed computing with clusters and library to write ML applications on top of it. Nevertheless, writing good Spark programs requires the user to understand the Spark session, data abstractions and transformations. Moreover, writing independent code for each ML application adds code redundancy. This paper enables end-users to create Spark ML applications by circumventing the tedious task of learning the Spark programming framework through a flow-based programming paradigm. Our main contributions include taking Java APIs of Spark ML operating on DataFrame, a popular ML library of Apache Spark, modelling them as composable components, and developing a conceptual approach to parse an ML flow created by connecting several such components. The conceptual approach has been validated by designing three ML use-cases involving prediction using decision trees, anomaly detection with k-means clustering, and collaborative filtering techniques to develop a music recommender application. The use-cases demonstrate how easily ML flows can be created graphically by connecting different components at a higher level of abstraction, parameters to various components being able to be configured with ease, automatic parsing of the user flow to give feedback to the user if a component has been used in a wrong position in a flow and finally automatic generation of ML application without the end-user having to

write any code. In addition to this, our work lays the foundation for several future works. This includes data visualisation techniques to make it more promising for the end-users to work on ML problems. Automatic deployment of the ML model with one click after the training phase would be another extension of our work. Design and implementation of a flow validation mechanism based on input/output of the flow component would make the system more flexible for future changes and generic for all kinds of flow design.

Author Contributions: Conceptualization, T.M.; Methodology, T.M.; Software, S.N.B.; Supervision, T.M.; Validation, S.N.B.; Writing—original draft, T.M. and S.N.B. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the German Research Foundation (DFG) and the Technical University of Munich within the Open Access Publishing Funding Programme.

Data Availability Statement: Not Applicable, the study does not report any data.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Zecevic, P.; Bonaci, M. Spark in Action. 2016. Available online: http://kingcall.oss-cn-hangzhou.aliyuncs.com/blog/pdf/Spark%20in%20Action30101603975704271.pdf (accessed on 12 November 2021).
2. Daniel, F.; Matera, M. *Mashups: Concepts, Models and Architectures*; Springer: Berlin/Heidelberg, Germany, 2014.
3. Meng, X.; Bradley, J.; Yavuz, B.; Sparks, E.; Venkataraman, S.; Liu, D.; Freeman, J.; Tsai, D.; Amde, M.; Owen, S.; et al. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.* **2016**, *17*, 1235–1241.
4. Zaharia, M.; Chowdhury, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Spark: Cluster Computing with Working Sets. In Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10), Boston, MA, USA, 22–25 June 2010; USENIX Association: Boston, MA, USA, 2010; p. 10.
5. Zaharia, M.; Xin, R.S.; Wendell, P.; Das, T.; Armbrust, M.; Dave, A.; Meng, X.; Rosen, J.; Venkataraman, S.; Franklin, M.J.; et al. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* **2016**, *59*, 56–65. [CrossRef]
6. Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauley, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing. In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12), San Jose, CA, USA, 25–27 April 2012; USENIX Association: Berkeley, CA, USA, 2012; p. 2.
7. Stahl, T.; Völter, M.; Czarnecki, K. *Model-Driven Software Development: Technology, Engineering, Management*; John Wiley & Sons, Inc.: Hoboken, NJ, USA, 2006.
8. LeCun, Y.; Bengio, Y.; Hinton, G. Deep Learning. *Nature* **2015**, *521*, 436–44. [CrossRef] [PubMed]
9. Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. TensorFlow: A system for large-scale machine learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA, USA, 2–4 November 2016; pp. 265–283.
10. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. Available online: tensorflow.org (accessed on 12 November 2021).
11. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*; Curran Associates, Inc.: Red Hook, NY, USA, 2019; pp. 8026–8037.
12. Carbone, P.; Katsifodimos, A.; Ewen, S.; Markl, V.; Haridi, S.; Tzoumas, K. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* **2015**, *38*, 28–38.
13. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-Learn: Machine Learning in Python. *J. Mach. Learn. Res.* **2011**, *12*, 2825–2830.
14. Collobert, R.; Bengio, S.; Mariéthoz, J. *Torch: A Modular Machine Learning Software Library*; Idiap-RR Idiap-RR-46-2002; IDIAP: Martigny, Switzerland, 2002.
15. Virtanen, P.; Gommers, R.; Oliphant, T.E.; Haberland, M.; Reddy, T.; Cournapeau, D.; Burovski, E.; Peterson, P.; Weckesser, W.; Bright, J.; et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nat. Methods* **2020**, *17*, 261–272. [CrossRef] [PubMed]
16. Culjak, I.; Abram, D.; Pribanic, T.; Dzapo, H.; Cifrek, M. A brief introduction to OpenCV. In Proceedings of the 35th International Convention MIPRO, Opatija, Croatia, 21–25 May 2012; pp. 1725–1730.
17. Nguyen, G.; Dlugolinsky, S.; Bobák, M.; Tran, V.; López García, Á.; Heredia, I.; Malík, P.; Hluchý, L. Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey. *Artif. Intell. Rev.* **2019**, *52*, 77–124. [CrossRef]
18. Morrison, J.P. *Flow-Based Programming: A New Approach to Application Development*, 2nd ed.; CreateSpace: Paramount, CA, USA, 2010.

19. Hall, M.; Frank, E.; Holmes, G.; Pfahringer, B.; Reutemann, P.; Witten, I.H. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.* **2009**, *11*, 10–18. [CrossRef]
20. Washington, M. *Azure Machine Learning Studio for The Non-Data Scientist: Learn How to Create Experiments, Operationalize Them Using Excel and Angular .Net Core . . . Programs to Improve Predictive Results*, 1st ed.; CreateSpace Independent Publishing Platform: North Charleston, SC, USA, 2017.
21. Berthold, M.R.; Cebron, N.; Dill, F.; Gabriel, T.R.; Kötter, T.; Meinl, T.; Ohl, P.; Thiel, K.; Wiswedel, B. KNIME—The Konstanz Information Miner: Version 2.0 and Beyond. *SIGKDD Explor. Newsl.* **2009**, *11*, 26–31. [CrossRef]
22. Demšar, J.; Curk, T.; Erjavec, A.; Črt Gorup; Hočevar, T.; Milutinovič, M.; Možina, M.; Polajnar, M.; Toplak, M.; Starič, A.; et al. Orange: Data Mining Toolbox in Python. *J. Mach. Learn. Res.* **2013**, *14*, 2349–2353.
23. BigML. Machine Learning That Works. 2020. Available online: https://static.bigml.com/pdf/BigML-Machine-Learning-Platform.pdf?ver=5b569df (accessed on 6 June 2020).
24. mljar. Machine Learning for Humans! Automated Machine Learning Platform. 2018. Available online: https://mljar.com (accessed on 18 May 2020).
25. Jannach, D.; Jugovac, M.; Lerche, L. Supporting the Design of Machine Learning Workflows with a Recommendation System. *ACM Trans. Interact. Intell. Syst.* **2016**, *6*, 1–35. [CrossRef]
26. StreamAnalytix. Self-Service Data Flow and Analytics For Apache Spark. 2018. Available online: https://www.streamanalytix.com (accessed on 18 May 2020).
27. Santos, W.d.; Avelar, G.P.; Ribeiro, M.H.; Guedes, D.; Meira, W., Jr. Scalable and Efficient Data Analytics and Mining with Lemonade. *Proc. VLDB Endow.* **2018**, *11*, 2070–2073. [CrossRef]
28. StreamSets. DataOps for Modern Data Integration. 2018. Available online: https://streamsets.com (accessed on 18 May 2020).
29. Armbrust, M.; Xin, R.S.; Lian, C.; Huai, Y.; Liu, D.; Bradley, J.K.; Meng, X.; Kaftan, T.; Franklin, M.J.; Ghodsi, A.; et al. Spark SQL: Relational Data Processing in Spark. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15), Melbourne, Australia, 31 May–4 June 2015; Association for Computing Machinery: New York, NY, USA, 2015; pp. 1383–1394. [CrossRef]
30. Walls, C. Spring Boot in Action. 2016. Available online: https://doc.lagout.org/programmation/Spring%20Boot%20in%20Action.pdf (accessed on 12 November 2021).
31. Freeman, A. *Pro Angular 6*, 3rd ed.; Apress: New York, NY, USA, 2018. Available online: https://link.springer.com/book/10.1007/978-1-4842-3649-9 (accessed on 12 November 2021).
32. Hajian, M. *Progressive Web Apps with Angular: Create Responsive, Fast and Reliable PWAs Using Angular*, 1st ed.; APress: New York, NY, USA, 2019.
33. Escott, K.R.; Noble, J. Design Patterns for Angular Hotdraw. In Proceedings of the 24th European Conference on Pattern Languages of Programs (EuroPLop'19), Irsee, Germany, 3–7 July 2019; Association for Computing Machinery: New York, NY, USA, 2019. [CrossRef]
34. Fowler, M. *Patterns of Enterprise Application Architecture*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2002.
35. JavaPoet. Available online: https://github.com/square/javapoet (accessed on 18 May 2020).
36. Overton, M.A. The IDAR Graph. *Queue* **2017**, *15*, 29–48. [CrossRef]
37. Overton, M.A. The IDAR Graph. *Commun. ACM* **2017**, *60*, 40–45. [CrossRef]
38. University of Irvine. UC Irvine Machine Learning Repository. Available online: https://archive.ics.uci.edu/ml/machine-learning-databases/covtype/ (accessed on 12 November 2021).
39. Mahapatra, T. High-Level Graphical Programming for Big Data Applications. Ph.D. Thesis, Technische Universität München, München, Germany, 2019.
40. Mahapatra, T.; Gerostathopoulos, I.; Prehofer, C.; Gore, S.G. Graphical Spark Programming in IoT Mashup Tools. In Proceedings of the 2018 Fifth International Conference on Internet of Things: Systems, Management and Security, Valencia, Spain, 15–18 October 2018; pp. 163–170. [CrossRef]
41. Mahapatra, T.; Prehofer, C. aFlux: Graphical flow-based data analytics. *Softw. Impacts* **2019**, *2*, 100007. [CrossRef]
42. Mahapatra, T.; Prehofer, C. Graphical Flow-based Spark Programming. *J. Big Data* **2020**, *7*, 4. [CrossRef]