UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

Improving Machine Learning Pipeline Creation using Visual Programming and Static Analysis

João Pedro Vieira David

Mestrado em Engenharia Informática
Especialização em Engenharia de Software

Dissertação orientada por:
Professor Doutor Alcides Miguel Cachulo Aguiar Fonseca

2021

Agradecimentos

Quero começar por agradecer ao Professor Alcides por me ter guiado, apoiado e desafiado ao longo deste ano de trabalho. A minha participação no projecto CAMELOT contribuiu significativamente para o meu desenvolvimento pessoal e profissional. Quero agradecer também a todos os meus colegas das reuniões semanais de trabalho pelo apoio e pelo feedback dado ao longo do ano, em especial ao Guilherme, pela sua participação mais activa no projecto.

Quero agradecer a todos os amigos, colegas e professores, tanto de licenciatura como de mestrado, por me terem acompanhado e auxiliado ao longo destes cinco anos de desafios.

Por fim, quero agradecer à minha família pela educação e apoio que deram ao longo da minha vida, foram imprescindíveis para me tornar a pessoa que sou hoje. Agradeço à minha avó, ao meu pai, à minha mãe, e à minha irmã.

João David

This work was supported by the Fundação para a Ciência e a Tecnologia (FCT) under LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020) and the CMU Portugal project CAMELOT (POCI-01-0247-FEDER-045915).

Dedicado à minha Avó

Resumo

A utilização de Machine Learning (ML) tem vindo a aumentar ao longo dos últimos anos, sendo utilizado em áreas tão distintas como a medicina de precisão, prevenção de fraude fiscal, optimização de compiladores, assistência médica e genética.

A aplicação de ML em qualquer domínio envolve uma sequência de tarefas que incluem extracção e pre-processamento de dados, e treino de modelos com posterior avaliação. Este fluxo de trabalho é denominado pipeline de ML e existem várias formas de criar e orquestrar estas pipelines.

A abordagem com base em código é a mais usada por data scientists (cientistas de dados). Requer que o programador trate de importar as bibliotecas de ML e ciência de dados necessárias, instancie objectos, passe parâmetros a funções e ainda trate de erros de compilação. Para além de ser necessário ter conhecimento de ML, os data scientists precisam ainda de ser programadores proficientes para poder entender e manter pipelines existentes. Estudos efectuados com data scientists demonstram que as suas maiores dificuldades estão no desenvolvimento de software. Encontrar pessoas que dominem tanto as áreas de data science como de engenharia de software poderá ser um desafio.

As ferramentas de programação visual para criação de pipelines de ML surgiram para remover a necessidade de um data scientist ter de saber programar para poder orquestrar pipelines. Através do uso de vários blocos que aplicam operações especificas aos dados e enviam o resultado para os blocos seguintes a que se encontrem ligados, o data scientist consegue criar pipelines de ML sem escrever uma única linha de código. Estas ferramentas proporcionam também uma experiência de criação de pipelines mais intuitiva, visto que o arrastar e ligar blocos transmite imediatamente o resultado final da pipeline, que pode ser discutido com relativa facilidade entre colegas de trabalho. No entanto, estes tipos de ferramentas têm algumas limitações que comprometem o seu uso, nomeadamente o facto de prenderem o utilizador. Quando uma pipeline de ML é construída, a sua continua manutenção tem de ser sempre feita usando a mesma ferramenta, sem que haja uma forma de migrar a pipeline para uma outra. Outro problema nestas ferramentas consiste no ambiente de desenvolvimento da pipeline ser o mesmo que o de execução. Após concluída a pipeline, a sua execução é feita dentro da ferramenta de desenvolvimento, impossibilitando assim uma execução mais eficaz num ambiente externo, como em clusters. O Orange é um exemplo de tais ferramentas.

Tanto a abordagem de código como a de programação visual para a criação de pipelines de ML apenas detectam erros na pipeline quando a parte problemática da pipeline é executada, levando

à perda de tempo e recursos para alcançar aquele ponto de execução. Isto é frequente quando se trabalha com datasets muito grandes que passam por várias operações de transformação de dados antes do erro ocorrer.

Este trabalho tem como objectivo melhorar sobre as lacunas existentes nas abordagens de orquestração de pipelines de ML, melhorando assim a sua experiência geral de construção. De forma a permitir a migração da pipeline pretendemos que esta seja compilada para uma linguagem de programação de uso geral onde a importação de bibliotecas é feita de forma automática, assim a pipeline pode ser mantida e alterada através da modificação do código fonte, permitindo ainda que seja executada em qualquer máquina (e.x., clusters). No entanto, de forma a prevenir que o data scientist fique limitado pelos blocos existentes na ferramenta de programação visual, propomos que a linguagem de programação visual seja extensível, permitindo que novos blocos possam ser implementados.

Em relação aos erros presentes na pipeline, em vez de serem detectados no momento da execução (como se verifica no Orange), pretendemos que sejam detectados de forma estática, sem que necessite de ser executada, permitindo aos data scientists perceber quais os impactos das suas decisões de desenho. Alguns exemplos de tais erros seriam a passagem de uma instância de um classificador em vez de um dataset, a passagem de um dataset desequilibrado a um classificador ou a passagem de um dataset que represente uma série temporal a um bloco para efectuar cross validation.

Este documento começa por apresentar de forma breve as bases de ML necessárias para compreender o conceito deste trabalho, nomeadamente em que consistem pipelines de ML e quais as fases que as constituem. Inicialmente existem fases de validação e pre-processamento de dados, onde são descartadas informações menos relevantes sobre o domínio do problema, ou são construídas informações mais relevantes. A estes procedimentos dá-se o nome de feature engineering. Depois de validados e trabalhados, os dados são usados na fase de treinamento e validação do modelo. Se o modelo passar na validação é então colocado num ambiente real a fazer previsões. Neste capítulo é ainda dada uma introdução a ML supervisionado com referência aos problemas de classificação e regressão.

De seguida apresenta o trabalho relacionado, onde foram estudadas as ferramentas existentes para orquestrar pipelines de ML através de programação textual e visual. Neste capítulo são ainda abordados trabalhos na literatura para verificar formalmente pipelines.

No capítulo seguinte é dada uma visão geral da implementação da ferramenta proposta neste trabalho para que o leitor seja primeiro introduzido à linguagem de programação visual para construir pipelines e compreenda os elementos essenciais de funcionamento da ferramenta, nomeadamente a forma como é feita a verificação estática das pipelines, antes de lhe ser apresentado os detalhes de implementação.

Posteriormente é detalhada de forma rigorosa o processo da implementação da ferramenta proposta neste trabalho, desde o front-end com a selecção das frameworks para o implementar como a sua arquitectura e os componentes que o constituem, até ao back-end onde corre o compilador

usado para verificar e compilar as pipelines criadas no front-end. Este capítulo explica também como é estabelecida a comunicação entre front-end e back-end, e como a ferramenta é enviada para produção.

De seguida é demonstrada a avaliação feita à ferramenta implementada. Visto que o seu objectivo é detectar problemas em pipelines de ML antes de serem executadas, a avaliação baseou-se na capacidade da ferramenta detectar erros comuns na construção de pipelines. Os erros usados foram identificados em reuniões com gestores de projetos de data science da Feedzai, de forma a garantir que a avaliação se focava em erros fundamentais e realistas. Por exemplo, concatenação de datasets com nomes iguais, criação de colunas com tipos incompatíveis, uso de datasets não balanceados, e ordem de operações na pipeline não optimizadas.

Por fim tiram-se conclusões sobre o trabalho realizado, cujas contribuições consistiram no desenho e implementação de uma linguagem de programação visual para construir pipelines de ML, no desenho e implementação de 29 blocos para efectuar operações de data science e ML, no desenho e implementação de um verificador de tipos para verificar de forma estática a correcção da pipeline, e por fim no desenho e implementação de um compilador que traduz a pipeline de uma linguagem de programação visual para código executável. É ainda mencionado as melhorias que podem ser feitas à ferramenta, nomeadamente o enriquecimento da linguagem com a implementação de mais blocos para programação visual, auto-completar novas ligações com base nos blocos compatíveis, optimizar a ordem das operações na pipeline de forma automática, e prever o tempo de execução da pipeline.

Palavras-chave: Programação Visual, Aprendizagem Automática, Pipeline, Verificação de Tipos, Compilador

Abstract

ML pipelines are composed of several steps that load data, clean it, process it, apply learning algorithms and produce either reports or deploy inference systems into production. In real-world scenarios, pipelines can take days, weeks, or months to train with large quantities of data. Unfortunately, current tools to design and orchestrate ML pipelines are oblivious to the semantics of each step, allowing developers to easily introduce errors when connecting two components that might not work together, either syntactically or semantically. Data scientists and engineers often find these bugs during or after the lengthy execution, which decreases their productivity.

We propose a Visual Programming Language (VPL) enriched with semantic constraints regarding the behavior of each component and a verification methodology that verifies entire pipelines to detect common ML bugs that existing visual and textual programming languages do not. We evaluate this methodology on a set of six bugs taken from a data science company focused on preventing financial fraud on big data. We were able detect these data engineering and data balancing bugs, as well as detect unnecessary computation in the pipelines.

Keywords: Visual Programming, Machine Learning, Pipeline, Verification, Data scientist

Contents

List of Figures

List of Tables

Code Listings

Acronyms

AI Artificial Intelligence.

API Application Programming Interface.

AST Abstract Syntax Tree.

DAG Directed Acyclic Graph.

DSL Domain Specific Language.

GPL General Purpose Language.

GUI Graphical User Interface.

HTTP HyperText Transfer Protocol.

IDE Integrated Development Environment.

JSON JavaScript Object Notation.

ML Machine Learning.

PCA Principal Component Anylisis.

PL Programming Language.

REST Representational State Transfer.

SMT Satisfiability Modulo Theories.

SVM Support Vector Machine.

UI User Interface.

VP Visual Programming.

VPL Visual Programming Language.

Chapter 1

Introduction

This thesis tackles the challenge of designing large-scale machine-learning pipelines. This chapter introduces the motivation for this work (Section 1.1), its goals (Section 1.2) and its contributions (Section 1.3).

1.1 Motivation

ML is becoming increasingly pervasive in the world, even with applications of which the user is unaware [1]. Examples of areas where ML usage is popular are precision medicine [2], financial fraud prevention [3], compiler optimization [4], healthcare [5] and even genetics [6].

The application of ML in any domain involves a sequence of tasks that include extracting and pre-processing data, model training, and model validation. This workflow is commonly referred to as ML pipeline. There are several ways to create and orchestrate these pipelines.

Data scientists commonly use a code-based approach. It requires the programmer to manually handle the imports of ML and data science libraries, instantiate objects, pass parameters to functions and debug compilation errors. In addition to having a ML background, data scientists [7] must also be proficient programmers, which could be a challenge for the development of ML pipelines due to the lack of skill they have in this domain.

Pereira, Cunha, and Fernandes [8] conducted interviews with eight participants in the field of data science to understand who are the data science workers of today, as well as the difficulties experienced by them and the technologies they use the most. For example, participant P8, who has no training in software engineering, reported his most significant challenge is developing stable and scalable code.

Kim et al. [9] presented a large-scale survey with 793 professional data scientists at Microsoft to understand their educational background, problem topics that they work on, tool usages, and activities. Of the respondents, only 24% identified as software engineers, software development engineers, or engineering managers.

Kery, Horvath, and Myers [10] used semi-structured interviews and a survey to study data scientists who program to experiment with data. The study showed that data scientists often code new analyses from previous ideas and use informal versioning to keep track of their code. For example, participant P6 even confessed to not caring about the quality of the code because he would not use it again.

The difficulty data scientists have with software engineering principles is the reason for the existence of tools that use Visual Programming (VP) to create pipelines since they remove the need for the data scientist to be a proficient programmer. VP tools have several blocks/widgets that apply specific operations to the data, forwarding the result to the following linked ones. However, these approaches have some limitations that compromise their use. For example, in tool lock-in, the user is forced to continue using the tool to maintain and improve the pipeline built, with no migration path to alternatives. Another limitation is that they do not support efficient, high-performance execution on clusters. Orange, which is an example of said tool, is explained in greater detail in Section 3.2 from the related work chapter.

Both code-based and VP approaches to build pipelines have a significant flaw with pipeline error detection. The errors are only detected at runtime when the problematic portion of the pipeline executes, wasting time and resources to reach that point. This problem often happens when working with large datasets with several data transformation operations before the error occurrence.

1.2 Goals

Driven by the gaps present in existing ML pipeline orchestrating approaches, this work aims to improve the overall experience of building ML pipelines. To tackle the tool lock-in issue, we propose to allow the compilation of the pipeline into a General Purpose Language (GPL), which uses specialized libraries for dataset manipulation and ML related algorithms. Thus, the pipeline would still be maintainable without using the tool by modifying the compiled source code. Furthermore, the tool's decoupling of the execution environment allows the pipeline to execute in any machine. Nevertheless, to prevent the data scientist from being limited by the blocks implemented in the VPL, the language would be extendable by implementing new custom ones.

Instead of approaching the pipeline creation as a continuous dynamic execution with every change triggering a re-run (like in the Orange tool), the approach would be to use dataset metadata to statically verify the pipeline correctness every time a change is made, without ever running the pipeline. A static verification would allow non-engineering data scientists to create ML pipelines and understand the run time performance impact of their design decisions by providing feedback and suggestions in the process of designing pipelines. We consider these type of errors as semantic errors, while other more syntactical errors would still be detected. An example of a more syntactical error (which would be a semantic error in a more general-purpose language like Python) would be the passing of a classifier instance instead of a dataset.

This work also proposes the detection of semantic errors, which novice data scientists often make. A program containing semantic errors executes until the end. Then, however, it uses the wrong methodologies to achieve an end goal. Examples of semantic errors are passing a dataset to a classifier without having the data balanced beforehand or cross-validating with a time series dataset. This work aims to detect and prevent both types of errors by having a static analysis checker within the pipeline creation environment.

1.3 Contributions

This work gives the following contributions:

- Design and implementation of a VPL approach to build ML pipelines
- Definition of a set of blocks for data science and Machine Learning operations
- Design and implementation of a type checker to statically verify pipeline semantic and syntactic correctness
- Design and implementation of a compiler to compile the VPL into executable code

1.4 Document Structure

This documents is organized as follows:

- Chapter 2 discusses the background required to understand the concept of this work
- Chapter 3 presents and discusses the related work
- Chapter 4 gives an overview of this work's implementation
- Chapter 5 presents the implementation of this work
- Chapter 6 evaluates the implemented work
- Chapter 7 draws conclusions

Chapter 2

Background

This chapter introduces the concept of ML pipelines and the phases they may contain, detailing Supervised Learning as an application example.

2.1 Machine Learning Pipelines

ML can be described as the usage of algorithms to automate the process of making a prediction based on existing data. Algorithms can take several steps to process data and prepare predictions. The organization of these steps composes a ML **pipeline**. The data processed throughout the pipeline is called **dataset**, which can consist of any kind. Images, audio, codebases, or tabular data. Figure 2.1 shows a high level view of a ML pipeline.

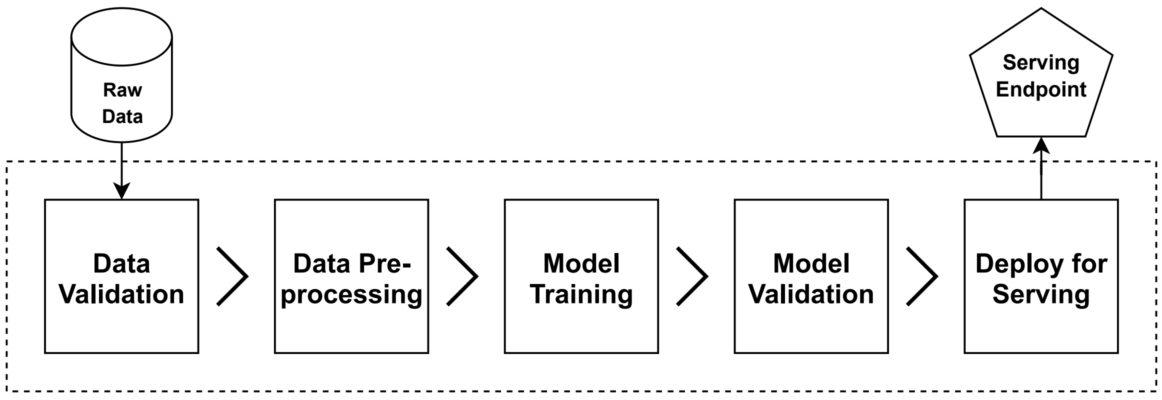


Figure 2.1: Phases of a ML Pipeline

Data Validation [11] is the process of verifying the quality of the data before feeding it to the ML pipeline. **Data Pre-processing** [12] is the process of applying several techniques to the data before the model learns from it, including data cleaning, data fusion, data reduction, and data transformation. **Model Training** is where the learning algorithm parses the data received from the previous phase and learns from it. **Model Validation** is where the previously trained model predicts unseen pre-labeled data to assess its accuracy. Finally, **Deploy for Serving** is where the model is deployed in a real-world environment, constantly parsing data and making predictions.

Machine Learning approaches [13, 14] can be considered **Supervised Learning** (Section 2.2) when it implies a task of predicting values (regression, subsection 2.2.2) or labels (classification, subsection 2.2.1) based on trusted pre-labeled data. **Unsupervised Learning** when it aims to extract knowledge from uncategorized data by clustering similar instances in groups. **Reinforcement Learning** when it aims to take suitable action to maximize reward in a particular situation. This thesis focuses on Supervisioned Learning.

2.2 Supervised Learning

In Supervised Learning, each instance in the dataset (presented as rows) has an associated target value (i.e., labeled data) along with a set of known and target features (presented as columns). Thus, Supervised Learning is the task of predicting target features from known features and requires a dataset that includes actual values for all features as a basis for learning.

The model trains using a dataset and afterward predicts the target value for new incoming data points (missing the target value). If the values consist of discrete class labels, then it is considered a classification problem (subsection 2.2.1). If instead, they are continuous values, then it is a regression problem (subsection 2.2.2). Figure 2.2 illustrates a Supervised Learning ML Pipeline. The pipeline begins by splitting the input dataset into two separate ones. The first is the training dataset used to train the model, while the second is the test dataset used to evaluate an already trained model. This evaluation consists of having the model predict the target value of unseen data points to compare with the actual value.

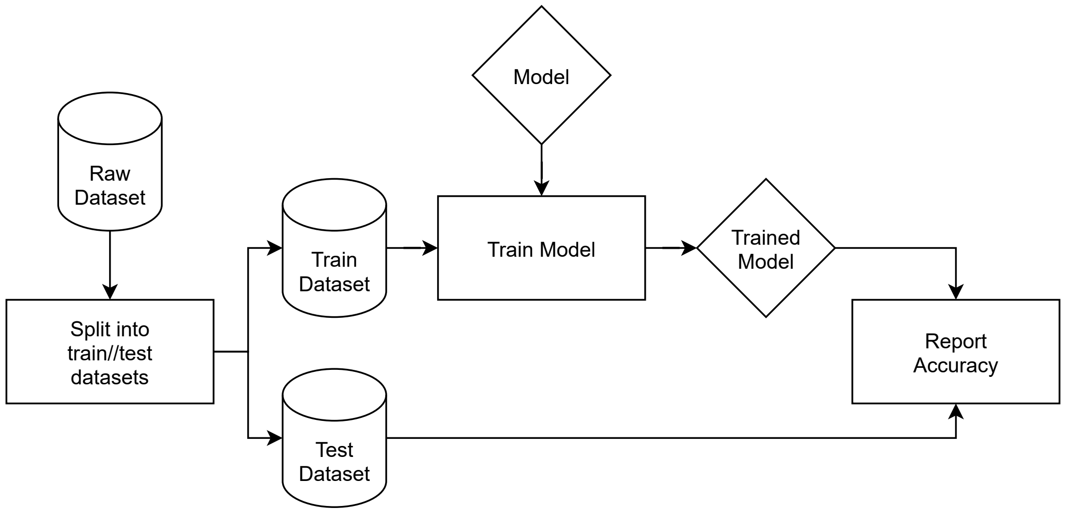


Figure 2.2: Supervised Learning ML Pipeline

2.2.1 Classification

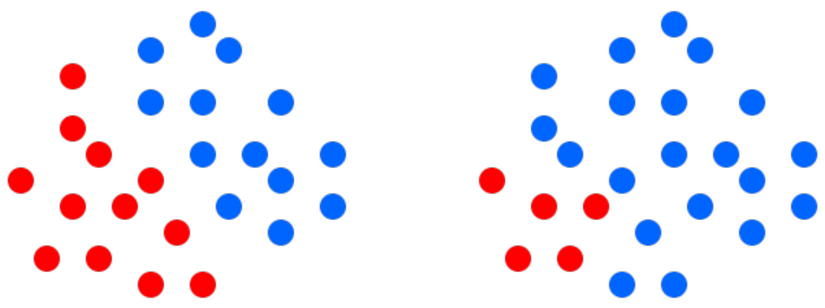
Classification [15] problems are Supervisioned Learning problems where the domain of possible values for the target column of the dataset consists of discrete class labels. Table 2.1 is a snippet of a credit card fraud dataset from Kaggle (`https://www.kaggle.com/mlg-ulb/creditcardfraud`). Features V1, V2, to V28 are the principal components obtained with Principal Component Anylisis (PCA), which is a technique to reduce the number of features. The only features which have not been transformed with PCA are **Time**, which contains the seconds elapsed between each transaction and the first transaction in the dataset, and **Amount**, which is the transaction amount. The **Class** column is the label for each transaction, and it takes value 1 in case of fraud and 0 otherwise.

Time	V1	...	V28	Amount	Class
0	-1.359807134		-0.021053053	149.62	0
0	1.191857111		0.014724169	2.69	0
1	-1.358354062		-0.059751841	378.66	0
1	-0.966271712		0.061457629	123.5	0
472	-3.043540624		0.035764225	529	1
4462	-2.303349568		-0.153028797	239,93	1

Table 2.1: Snippet of the Credit Card Fraud Dataset

Most ML datasets have thousands or millions of records. The size and quality of the dataset correlate with the difficulty of collecting trustful information about the problem in question and its complexity. For very complex problems, there is the need to have several data points (records) and several features (columns) describing the properties of each instance. This complexity can also bring other issues, such as data skewness.

When records for some class label are more accessible to obtain than other class labels within the problem domain, the dataset is considered an **Imbalanced Dataset**, meaning that the total number of each label is highly disproportionate, which is the case on the credit card fraud dataset, where the majority of transactions not being fraudulent. On the other hand, when the proportions are similar, it is considered a **Balanced Dataset**. Figure 2.3 demonstrates the proportions of each dataset, where each point of the same color represents a data point with the same class label.



(a) Balanced Dataset (b) Imbalanced Dataset

Figure 2.3: Balanced/Imbalanced Datasets Label Proportions

Models are ideally trained on balanced datasets and validated on unseen datasets, which do not necessarily need to be balanced. Nonetheless, it must have enough records from each target variable to increase the confidence levels in the model, since learning with more instances typically results in better performance, especially regarding the under-represented class [16].

When selecting a classifier for a pipeline, it is crucial to take into account its characteristics:

- **training time**, the time necessary to train a classifier, which is the process of mapping input data (the features) to an output value (the target values);
- **inference time**, the time for a trained classifier to make a prediction;
- **generalization ability**, the performance of a trained classifier on predicting unseen data.

There are two types of learners in classification [15], differing on the training process:

- **Lazy learners**, those that store the training data, and use it when then faced with the task of predicting a target value for an incoming input. They have a small training time (dependent on the time needed to store the data) and a big inference time since it needs to parse the stored data to predict an outcome;
- **Eager learners**, which construct a classification model based on the given training data, discarding the data afterward. Due to the model construction, eager learners have a considerably longer training time when compared to lazy learners. However, the inference time is short.

For instance, the K-Nearest Neighbors classifier spends little time on the training phase since the core idea behind this algorithm is to find the K (a parameter) nearest neighbors to the instance being predicted. The neighbor's target majority will dictate the label prediction. Therefore it is considered a lazy learner classifier.

On the other hand, Naive Bayes, Support Vector Machines (SVM), Decision Trees, and Random Forests take longer during the training phase, and then each prediction is much faster. Therefore they are considered eager learners. It is clear to see that, for a problem where the predictions must be quick, the K-Nearest Neighbors is not ideal. There are several other classifier algorithms, each one of them performs the training in different ways. Depending on the datasets and problem, one may choose different classifiers.

2.2.2 Regression

In regression problems, the domain of possible values for the label column of the dataset are continuous quantities. The dataset snippet shown previously could be adopted to a regression problem by changing the label column from a binary label to the probability of a transaction being fraudulent or not. Table 2.2 is an example of a regression dataset. The predictions of a trained regressor consist of predicting probability values in the [0;1] threshold.

Time	V1	...	V28	Amount	Fraudulent Score
0	-1.359807134		-0.021053053	149.62	0.54
0	1.191857111		0.014724169	2.69	0.03
1	-1.358354062		-0.059751841	378.66	0.13
1	-0.966271712		0.061457629	123.5	0.32
472	-3.043540624		0.035764225	529	0.94
4462	-2.303349568		-0.153028797	239,93	0.96

Table 2.2: Regression dataset

Chapter 3

Related Work

This chapter introduces popular libraries used to implement ML projects, several ML orchestration frameworks that aid data scientists when creating complex pipelines, and how the correctness of pipelines is verified.

3.1 Machine Learning Libraries

Data scientists frequently rely on ML libraries and orchestration frameworks. Libraries provide off-the-shelf transformations, models, and metrics, while orchestration frameworks allow a high-level definition of the pipeline and automate its low-level execution.

3.1.1 General-purpose Libraries

One set of ML libraries are those that allow the creation of a project end-to-end, providing algorithms for each phase of the ML process.

scikit-learn Scikit-learn [17] is an open-source Python library for ML, containing many classification, regression, clustering, and dimensionality reduction algorithms useful for solving supervised and unsupervised learning problems. It is available at `https://github.com/scikit-learn/scikit-learn`.

Weka Weka [18] is an open-source ML software widely used for teaching, research, and industrial applications. The ML pipelines can either be built using a Graphical User Interface (GUI) or using its Java API. The former does not require programming, while the latter does, similarly to Scikit-learn. Both methods come with implemented classifiers such as RandomForest and NaiveBayes. It is available at `https://github.com/Waikato/weka-3.8`.

H2O H2O [19] is an open-source ML platform with linear scalability, supporting the most widely used statistical and ML algorithms, including gradient boosted machines, generalized linear models, and deep learning. It is mostly used in enterprise scenarios, and it works on existing big data infrastructures, bare metal, or on top of existing Hadoop, Spark, or Kubernetes clusters. Some key features of H2O are the ability to use other Programming Language (PL) such as R and Python to build models. It is available at `https://github.com/h2oai/h2o-3`.

3.1.2 Specific-purpose Libraries

Other ML libraries are specific for tasks in a single ML phase.

TensorFlow TensorFlow [20] is a complex library for distributed numerical computation using data flow graphs, allowing the training and running of extensive neural networks efficiently by distributing the computations across potentially thousands of multi-GPU servers. TensorFlow was created at Google and later open-sourced in 2015, and it is available at `https://github.com/tensorflow/tensorflow`.

XGBoost XGBoost [21] is an optimized distributed gradient boosting library designed to be highly efficient, flexible, and portable. It implements ML algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solves many data science problems quickly and accurately. The same code runs on major distributed environments (Kubernetes, Hadoop, SGE, MPI, Dask) and can solve problems beyond billions of examples. Its implementation is available at `https://github.com/dmlc/xgboost`.

3.2 Machine Learning Orchestration

When implementing a ML pipeline, it is common to do it manually by implementing several methods with specific tasks, such as data splitting or model training. Then they are called with a specific order to create a flow of data to implement the actual pipeline. This approach is not ideal in complex ML projects where it is common to have new people joining in. Enterprise projects need to have an organized development environment so that every team member, especially newcomers, understands the pipeline structure without spending days looking through several source files.

ML orchestration frameworks deliver improved development environments by having well-defined separation of concerns that provide better code structure. Therefore, teams can easily discuss pipeline structure and data flow while avoiding commit conflicts when further improving the pipeline. The following paragraphs present the ML orchestrations frameworks studied during this thesis.

igel igel [22] is built on top of the scikit-learn and pandas libraries for the ML and data manipulation algorithms, respectively. The user defines the pipeline configuration from a file and then runs it through the command line. The authors of this tool also developed a simple User Interface (UI) that uses igel on the back-end. Its implementation is available at `https://github.com/nidhaloff/igel`.

Kedro Kedro [23] is an open-source Python framework that applies software engineering best practices to data and ML pipelines by breaking large chunks of code into small modular units (`https://github.com/quantumblacklabs/kedro`).

The data catalog, which consists of a YAML file used to reference datasets, handles the loading of datasets that can be stored locally or in the cloud by giving them a variable name.

Each ML pipeline operation unit is considered a Node, which consists of a Python function that receives data as input from previous Nodes and returns the resulting data to be used by other Nodes. The implementation of each operation in individual Nodes allows for the separation of concerns within the project.

The pipeline can then be seen as a DAG composed by the defined nodes by using Kedro-Viz (`https://github.com/quantumblacklabs/kedro-viz`). Kedro-Viz is a plugin that generates a visual representation of the pipeline directly from the code, providing a high-level overview of the data pipeline structure, which could be used to communicate the workflow of the pipeline with colleagues and stakeholders, and give a better insight into the inner working of the pipeline to new colleagues joining the project.

Dagster Dagster [24] is an open-source Python data orchestrator for ML that allows the definition of pipelines in terms of data flow between reusable and logical components. The pipelines can then be tested locally and deployed anywhere. Dagster can schedule and orchestrate pandas, Spark, SQL, or any other Python library. It comes with Dagit, a web interface debugger that allows for the inspection of executed pipelines, where it is possible to query over logs, discover the most time-consuming tasks via a Gantt chart, and re-execute subsets of steps. Dagster is designed for data platform engineers, data engineers, and full-stack data scientists. Available at `https://github.com/dagster-io/dagster`.

Orange Orange [25] is an open-source data visualization, ML and data mining toolkit that uses scikit-learn and pandas libraries for the ML and data manipulation algorithms. It features a VP environment, where the user can create ML pipelines by connecting series of widgets. Available at `https://github.com/biolab/orange3`.

Figure 3.1 is a screenshot of the orange IDE. The sidebar at the left contains all the available widgets for the data scientist to use, while at the right is where the creation of the pipeline happens by connecting widgets.

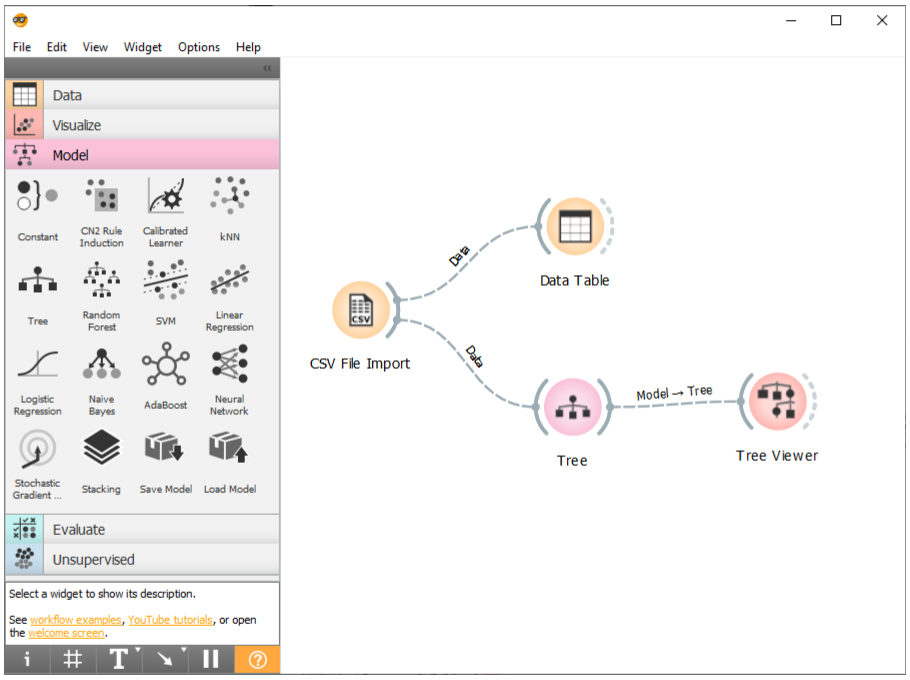


Figure 3.1: Orange IDE

KNIME Analytics Platform KNIME Analytics Platform [26] is an open-source end-to-end data science software for the creation of ML pipelines using VP. It features a drag-n-drop UI to create blocks and links without the need for coding. Features several data sources, ranging from the common local CSV file to connecting to hosts of databases such as Oracle, Microsoft SQL, and Apache Hive. It also allows data retrieval from sources such as Salesforce, SharePoint, SAP Reader (Theobald), Twitter, AWS S3, Google Sheets, and Azure. Available at `https://www.knime.com`.

Datrics Datrics [27] is an end-to-end data science for the creation of ML pipelines. The data scientist drags and links blocks to build the pipeline without the need for coding, allowing for the processing of data, as well as training and deployment of ML models. It is available at `https://datrics.ai`.

3.3 Formal Methods for Machine Learning

Formal methods are an array of techniques that employ logic and mathematics to provide rigorous guarantees about the correctness of computer software. They are an integral part of the development process of traditional (non-machine-learned) critical software systems to provide robust, mathematically grounded guarantees on the software behavior. For instance, they are used at an industrial level in avionics [28], where the development processes of aircraft software systems have very stringent assurance and verification requirements mandated by international standards.

Advances in Artificial Intelligence (AI) and ML along with the availability of vast amounts of data, allows for the development of computer software that efficiently and autonomously performs complex tasks that are difficult or even impossible to design using traditional explicit programming (e.g., image classification, speech recognition). This makes ML desirable in many applications, including safety-critical applications. For instance, in the avionics industry, ML is used for image-based operations (taxiing, takeoff, landing) and aircraft voice control, while in the automotive industry, it is used for autonomous driving.

The following paragraphs address current advances in formal methods for ML models, as well as data preparation, the focus of this work [29].

Neural Networks Kurd and Kelly[30] proposed a characterization of verification goals for neural networks used in safety-critical applications. Most formal methods for neural networks aim at verifying what they identified as goals G4 and G5. G4 states that neural networks are robust and safe under all input conditions, while G5 states that the output should not be hazardous regardless of the integrity of the input.

Support Vector Machines Ranzato and Zanella [31] proposed an approach that focuses on providing local robustness to adversarial perturbations of Support Vector Machine (SVM) based on the most commonly used kernel functions (linear, polynomial, and radial basis function kernels). It is implemented and available in an open-source tool named **SAVer** (`https://github.com/abstract-machine-learning/saver`).

Decision Tree Ensambles Kantchelian et al. [32] proposed an approach for finding the nearest adversarial example concerning the L0, L1, L2, and $L\infty$ distances.

Chen et al. [33] presented a linear time algorithm for finding the nearest adversarial for a decision tree.

Sato et al. [34] proposed an SMT-based approach for safety verification of random forests and gradient boosted decision trees. Another SMT-based approach by Einziger et al. [35] verifies the local robustness to adversarial perturbations of gradient boosted decision trees used for classification.

Törnblom and Nadjm-Tehrani proposed an abstraction-refinement approach for random forests [36] and gradient boosted decision trees [37, 38] with univariate split predicates. It is implemented in an open-source tool named **VoTE** (`https://github.com/john-tornblom/VoTE`).

Ranzato and Zanella [39] proposed a more general framework that supports arbitrary abstract domains. The approach is available and implemented in an open-source tool named **Silva** (`https://github.com/abstract-machine-learning/silva`).

Calzavara et al. [40] have proposed an approach that supports perturbations modeled as arbitrary imperative programs (as opposed to distance-based perturbations).

Data Preparation Urban and Muller [41] proposed an abstract interpretation framework for reasoning about data usage and a static analysis method for automatically detecting (possibly accidentally) unused input data. Urban [42] also proposed a configurable static analysis for automatically inferring assumptions on the input data.

The overview of the state of the art shows that there are still no approaches that focus on verifying entire ML pipelines, which is necessary to ensure the safe use of ML software in safety-critical applications [29].

Chapter 4

Approach

The increasing popularity of ML over the years lead to a scarcity of data scientists proficient in programming and with good knowledge of data science.

ML pipeline orchestration can either be approached through code, or VP. The former requires the data scientist to manually handle the imports of ML and data science libraries and instantiate objects. The latter relies on having the data scientist drag-n-drop blocks and then have them linked to applying specific operations to the data received from previous blocks, forwarding the result to the following ones.

VP approaches to orchestrate ML pipelines originated from the lack of data scientists that are good at programming. By providing a more user-friendly approach to building pipelines, data scientists can focus more on data science and less on software engineering.

Nonetheless, existing VP approaches still have gaps. This chapter describes the general approach of the proposed tool and how it improves over existing ones.

4.1 Decoupled Visual Programming

Existing VP tools for building ML pipelines approach pipeline creation from a dynamic point of view. Datasets are fully loaded into the tool so that the pipeline re-executes as the data scientist modifies it. At first sight, the dynamic execution of the pipeline may seem excellent since it provides real-time feedback on the changes the datasets go through along the pipeline. However, when working with large datasets on complex pipelines that contain heavy data processing operations, the constant re-executions of the pipeline consumes time. Besides, having to wait for the pipeline to finish the execution constantly is unproductive. Moreover, forcing the pipeline to only execute inside the tool prevents it from being executed in more powerful machines, such as clusters, which is the case in enterprise ML pipelines.

Our proposed tool improves over existing ones by separating the pipeline's creation and execution environments. Instead of having a dynamic execution of the pipeline as it changes and requiring it to execute inside the tool, our approach features a compiler that translates the pipeline into an executable code. The code can then execute in a cluster for increased performance and better handling of big data. The advantage of using a compiler instead of an interpreter, like the one used in the Orange tool, is the possibility for optimizing the pipeline's DAG representation to have a parallel execution of the program.

4.2 Language Definition

Each block contains a set of input and output ports used to link to other blocks. Links are only possible between input and output ports of the same type, belonging to different blocks. Each block processes the data received in the input ports, forwarding the result to the output port. The user can adjust the block's properties to tune the block operation.

Instantiated blocks have a unique identifier that identifies them within the program. The block id and respective port id within the block from where the link starts (source ids) and the block and port id where the link ends (target ids) characterize a link, which also possesses a unique identifier in the program. The set of blocks and links formed between them characterize a program.

Listing 4.1 summarize the language of the blocks. Figure 4.1 presents the existing types and Figure 4.2 presents their relationships, following the syntax of Pierce [43].

```
MetaBlock
   input_ports_types: Dictionary<String,Type>
   properties_types: Dictionary<String,Type>
   output_ports_types: Dictionary<String,Type>
   assertions: List<Formula>

Block
   id: String
   name: String
   kind: MetaBlock
   properties: Dictionary<String,Object>
   input_ports: Dictionary<String,Port>
   output_ports: Dictionary<String,Port>

Link
   id: String
   source: Tuple<block_id:String, port_id:String>
   target: Tuple<block_id:String, port_id:String>

Program
   blocks: Dictionary<String,Block>
   links: Dictionary<String,Link>
```

Code Listing 4.1: Formal definition of block's language

$$\text{Types} \quad T ::= Object \mid Port \mid DataSet \mid Classifier \mid Regressor \mid Int \mid String \mid Float \mid Bool$$

Figure 4.1: The types used in the formal definition

$$\frac{}{T <: Object} \qquad \frac{}{DataSet <: Port} \qquad \frac{}{Classifier <: Port} \qquad \frac{}{Regressor <: Port}$$

(type-equality)

Figure 4.2: Relationships between types

For a program to be correct, the following conditions must be true.

Block correctness
For any b block:

- the set of keys in b.properties must be the same as in b.kind.properties_types
  - for any key k:
    - b.properties[k] must have the type in b.kind.properties_types[k]
- the set of keys in b.input_ports must be the same as in b.kind.input_ports_types
  - for any key k:
    - b.input_ports[k] must have the type in b.kind.input_ports_types[k]
- the set of keys in b.output_ports must be the same as in b.kind.output_ports_types
  - for any key k:
    - b.output_ports[k] must have the type in b.kind.output_ports_types[k]

Link correctness
For any program p:
For any link l in values(p.links):
For any b1 block:
For any b2 block:

- b1 != b2
- b1.id = l.source.block_id
- b2.id = l.target.block_id

then:

- b1.kind.output_ports_types[l.source.port_id] = b2.kind.input_ports_types[l.target.port_id]

Assertions verification
For each program p:

- Ab = { formula | formula ← b.kind.assertions, b ← values(p.blocks) }
- Al = { p.blocks[l.source.block_id].output_ports[l.source.port_id] = p.blocks[l.target.block_id].input_ports[l.target.port_id] | l ← keys(p.links) }

Set of formulas to satisfy = Ab ∪ Al
Smt_valid(Ab ∪ Al) must be true.

All ports are linked verification
For any program p:
For any link l in values(p.links):
For any b block in values(p.blocks):
For any in_p_key port in keys(b.input_ports) there is:

- l.source.port_id = in_p_key
- l.source.block_id = b.id
- l.id in p.links

The conditions presented here are only used to verify the program's correctness before compilation. In the program development phase, they are omitted.

4.3 Static Analysis for Machine Learning Pipelines

Every block type in our tool contains a set of template assertions. Some of them are hardcoded into the block's type implementation and shared by all instances of a block type. Others have placeholders to insert the values of the properties for the block instance and the dataset metadata (column names and types) passing through it.

Links between block instances contain their own set of assertions reflecting the passage of data between them. The assertions ensure dataset metadata equality between the metadata exiting one block and entering the following one.

Figure 4.3 exemplifies the assertions between two blocks and the link that connects them. Assuming that is used a dataset with 31 columns and 124 rows, those values replace the placeholders (underlined) for the *Import Dataset* block's assertions. The *Random Forest Classifier* block's assertions are hardcoded into the block type implementation. Therefore, all block instances of that type expect to receive a dataset with more than one column and more than zero rows. The link assertion establishes that the metadata of the dataset exiting A1 is the same entering B1.

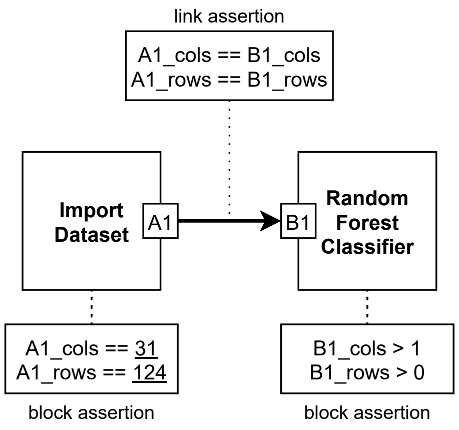


Figure 4.3: Block and link assertions

The collection of all block and link assertions in a given pipeline is added into an Satisfiability Modulo Theories (SMT) solver to verify the pipeline correctness. If the SMT solver returns **sat** no assertions are violated. If instead, it returns **unsat**, there are contradicting assertions within the solver that need to be identified. Subsection 5.3.4 describes the type-checking algorithm implemented to identify those assertions.

The assertions describe the dataset metadata along the pipeline. Examples of those properties are:

- Number of columns/rows
- Whether the dataset is balanced
- Whether the dataset represents a time series
- Whether the dataset has been reduced or processed

The programmer of blocks can add more properties to the existing ones to enrich the pipeline's type checking.

Chapter 5

Implementation

This chapter presents MLVP, a concretization of the approach presented in the previous chapter. We present its architecture (Section 5.1), the pipeline editor (Section 5.2), the compiler (Section 5.3) and the deployment strategy (Section 5.5).

5.1 Architecture

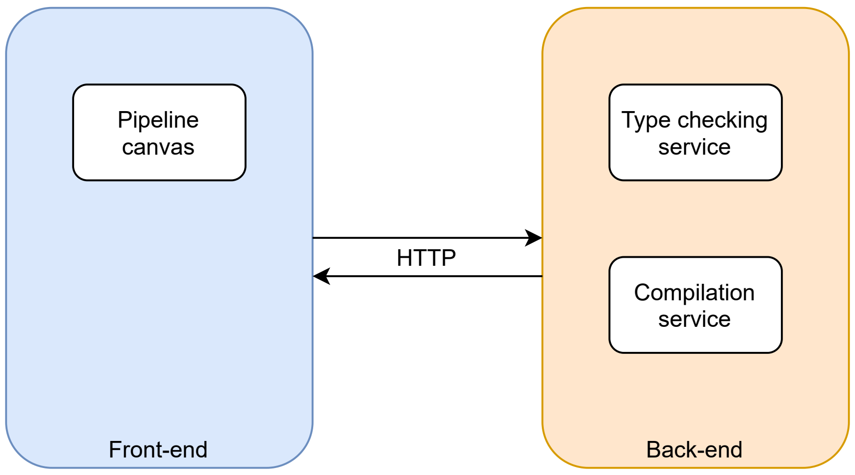


Figure 5.1: Architecture overview

MLVP is composed of a front-end and a back-end. The front-end contains the pipeline canvas where the data scientist creates ML pipelines by dragging and linking blocks (widgets in other tools) together. Pipeline changes trigger type checking requests to the back-end server that validates the semantic and syntactic correctness of the pipeline. The back-end server also handles the compilation of correct pipelines. The following sections explain in greater detail the inner workings of both front-end and back-end implementations.

5.2 Pipeline Editor

The pipeline editor is responsible for allowing the user to design and edit ML pipelines.

5.2.1 UI Library Selection

To implement a pipeline editor, we selected a framework that supports the design of pipelines using different blocks with multiple ports (the linking point between blocks). Each framework was evaluated by either implementing a straightforward project or using the samples implemented by the framework's authors. This evaluation allowed us to understand how each framework implements blocks and ports and restricts links between different ports.

The criteria considered most relevant when evaluating the diagram frameworks were the following:

1. Supports TypeScript [44]
2. Supports React [45]
3. Support different types of blocks and ports
4. Complexity of the links between nodes
5. Allowing to save and restore the canvas state
6. Active community using the framework

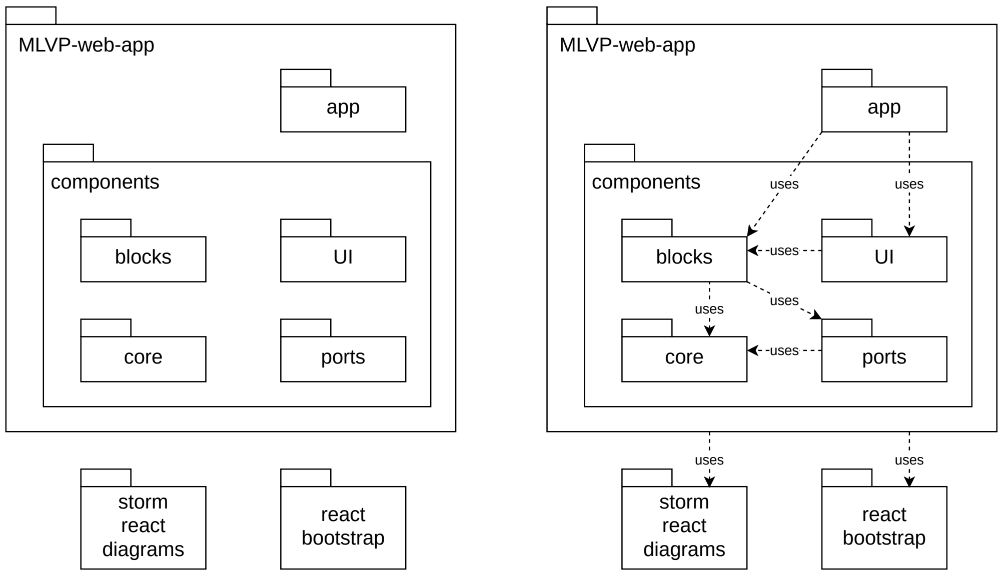
Name	**Coded in**	**Stars**	**Commits**	**Open/Closed Issues**	**npm Weekly Downloads**
Rete 1.4.5-rc.1 [46]	TypeScript	6.3k	475	63/393	4.8k
Beautiful React Diagrams 0.5.1 [47]	React JavaScript	2.1k	89	27/17	0.5k
Storm React Diagrams 6.3.0 [48]	React TypeScript	5.6k	762	191/444	7.2k

Table 5.1: Diagram frameworks Github stats

Table 5.1 presents the frameworks considered for this approach. Because of requirement 2, motivated by the increasing popularity of React, Rete was excluded.

A significant factor taken into consideration when choosing between the React frameworks was the engagement of the community on the GitHub repositories. Stats from Table 5.1 show that Storm React Diagrams had more popularity among users. The advantage of using a more popular framework is that if any problem or doubt about the framework arises, it is easier to resolve by looking through older issues. Therefore, the framework chosen to implement the project was the **Storm React Diagram** framework.

5.2.2 Front-end Design



(a) Decomposition View (b) Uses View

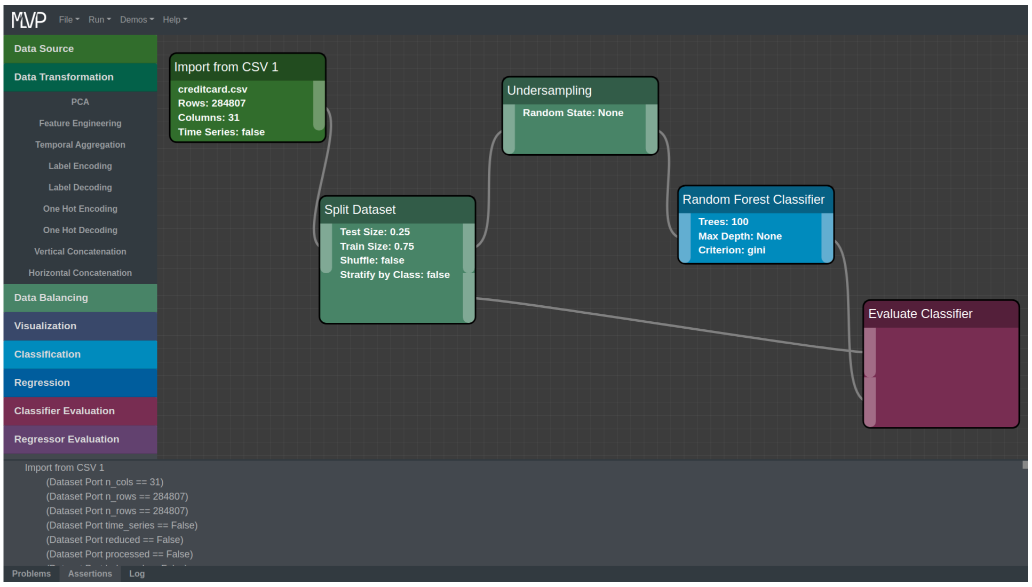
Figure 5.2: Web App Decomposition/Uses Module Views

Figures 5.2a and 5.2b respectively represent the Module Decomposition and Module Uses Views of the React web application. The ***app*** module contains the class component managing the overall app state, along with auxiliary classes defining the canvas responsiveness, for instance, for when a user zooms or drags and selects links. The ***components*** module, which contains four other sub-modules. The ***core*** sub-module contains the implementation for the base properties for the blocks and ports. The ***blocks*** sub-module contains as many sub-modules as block types, each sub-module containing the implementation for that respective block type. The ***UI*** contains the implementation for the canvas, app modal, top, bottom, and sidebar functional components. Finally, the ***ports*** module contains the several port types used by the blocks to create links between each other.

5.2.3 Pipeline Canvas

The pipeline canvas is the area in which the data scientist creates ML pipelines by dragging and linking blocks together. Each block receives and processes the data from its linked predecessors and then forwards it to its successors, thus recreating the base idea behind a ML pipeline, introduced in Section 2.1.

Figure 5.3 shows the MLVP tool. The sidebar on the left has eight differently coloured categories of blocks, each containing its block types. Dragging a block type into the canvas instantiates a block of said type. The pipeline in the canvas is composed of five blocks, from left to right, the first imports the dataset, the second splits it into train and test datasets, which will be then sent respectively to the *Random Forest Classifier* and *Evaluate Classifier* blocks. The former uses it to train the classifier, and the latter to evaluate the classifier after training.



creditcard.csv
Rows: 284807
Columns: 31
Time Series: false

Random State: None

Test Size: 0.25
Train Size: 0.75
Shuffle: false
Stratify by Class: false

Trees: 100
Max Depth: None
Criterion: gini

```
Import from CSV 1
    (Dataset Port n_cols == 31)
    (Dataset Port n_rows == 284807)
    (Dataset Port n_rows == 284807)
    (Dataset Port time_series == False)
    (Dataset Port reduced == False)
    (Dataset Port processed == False)
```

Figure 5.3: MLVP

Blocks are linked through their ports and can contain only input ports (on the left), only output ports (on the right), or both. Regardless of the case, links are created between input and output ports of the same type from different blocks.

5.2.4 Pipeline Serialization

The canvas is serialized into a JavaScript Object Notation (JSON) for two purposes: to save the pipeline within for later and to communicate the state of the pipeline to the back-end server for semantic error detection and compilation. This JSON is constituted by several key/value pairs represented in Listing 5.1. Its structure is crucial to how the compiler works. The blocks and links from Figure 5.4 will serve as running example throughout this section.

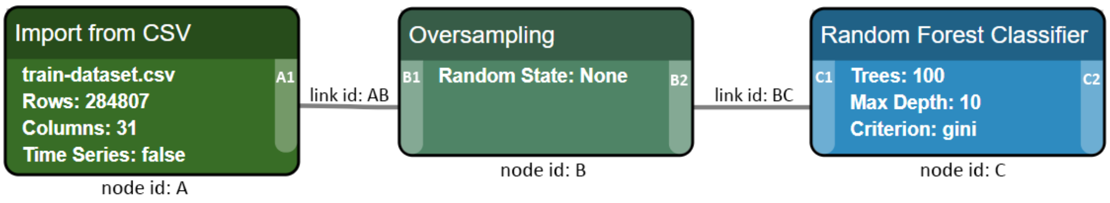


Figure 5.4: Block links used for JSON structure explanation

Starting with the *layers* key, which contains the ***diagram-links*** and ***diagram-nodes*** layers, the former containing all links established between blocks, the latter the blocks themselves.

```
{
   "id": "d3449e70-7290-4d05-bbfa-fe6290b810a6",
   "offsetX": -104.4161484859954,
   "offsetY": 136.31951963396432,
   "zoom": 125,
   "gridSize": 0,
   "layers": [ {diagram-links}, {diagram-nodes} ]
}
```

Code Listing 5.1: JSON structure

```
"type": "diagram-links",
"models": {
   "AB": {
      "id": "AB",
      "source": "A",
      "sourcePort": "A1",
      "target": "B",
      "targetPort": "B1",
   },
   "BC": {
      "id": "BC",
      "source": "B",
      "sourcePort": "B2",
      "target": "C",
      "targetPort": "C1",
   },
}
```

Code Listing 5.2: diagram-links layer

Let us consider the ***diagram-links*** layer (Listing 5.2), which has two link models in total, the first being between blocks *Import from CSV* and *Oversampling*, and the second from *Oversampling* to *Random Forest Classifier*. Every link model is defined by having its key, the id within the diagram, and the value, a JSON object with the corresponding data regarding that link.

The ***source*** property stores the id of the block originating the link, the ***sourcePort*** property, the port id within that block establishing the link (a block can have multiple ports, this way it is possible to distinguish them). Similarly, the ***target*** and ***targetPort*** store the block and port id at the other end of the link. Since links can only be created from an output port to an input port of two distinct blocks, the ***sourcePort*** is always an output port, and the ***targetPort*** an input port.

Regarding the ***diagram-nodes*** layer (shown in Listing 5.3, omitting the ports array from all blocks except *Oversampling*, for brevity), every block model shares a common set of properties, such as ***id***, ***type***, ***ports*** and ***title***, being in the same order, the id of the block, the unique name type as well as the back-end class representing it, the array of ports within the block and the block's title.

```
  "type": "diagram-nodes",
  "models": {
     "A": {
        "type": "ImportFromCSV",
        "ports": [ {"id": "A1"} ],
        "fileName": "train-dataset.csv",
        "numCols": 5,
        "numRows": 150,
     },
     "B": {
        "id": "B",
        "type": "Oversampling",
        "ports": [{
              "id": "B1",
              "type": "DatasetPort",
              "name": "Dataset",
              "parentNode": "B",
              "links": [
                 "AB"
              ],
              "in": true,
           },
           {
              "id": "B2",
              "type": "DatasetPort",
              "name": "Balanced Dataset",
              "parentNode": "B",
              "links": [
                 "BC"
              ],
              "in": false,
           }
        ],
        "randomState": None
     },
     "C": {
        "type": "RandomForestClassifier",
        "ports": [ {"id": "C1"}, {"id": "C2"} ],
        "numTrees": 100,
        "criterion": "gini",
        "maxDepth": 10
     }
  }
```

Code Listing 5.3: diagram-nodes layer

Each block type also has its respective set of properties regarding the logic behind it. For instance, the *Import From CSV* block has the ***fileName***, ***numCols*** and ***numRows*** properties, describing the information about the dataset imported, and the *Random Forest Classifier* block the ***numTrees***, ***criterion*** and ***maxDepth*** properties to set the classifier settings.

Each port model also has its own properties, ***id***, ***type***, ***name***, ***parentNode*** and ***in***, respectively they represent its id, its type (links can only be created between ports of the same type) as well as the back-end class that represents it, the name of the port (useful to distinguish data flow when a block has one or more ports of the same type), the id of the block owning the port, and whether it is an input port or not.

5.3 Compiler

The compiler is responsible for verifying the pipeline and generating the final code that will execute it. This section describes in depth the several phases of the compiler, the Parser (subsection 5.3.1), the Topological Sorter (subsection 5.3.2), the Data Flow Elaboration (subsection 5.3.3), the Type Checker (subsection 5.3.4) and finally the Code Generator (subsection 5.3.5).

5.3.1 Parser

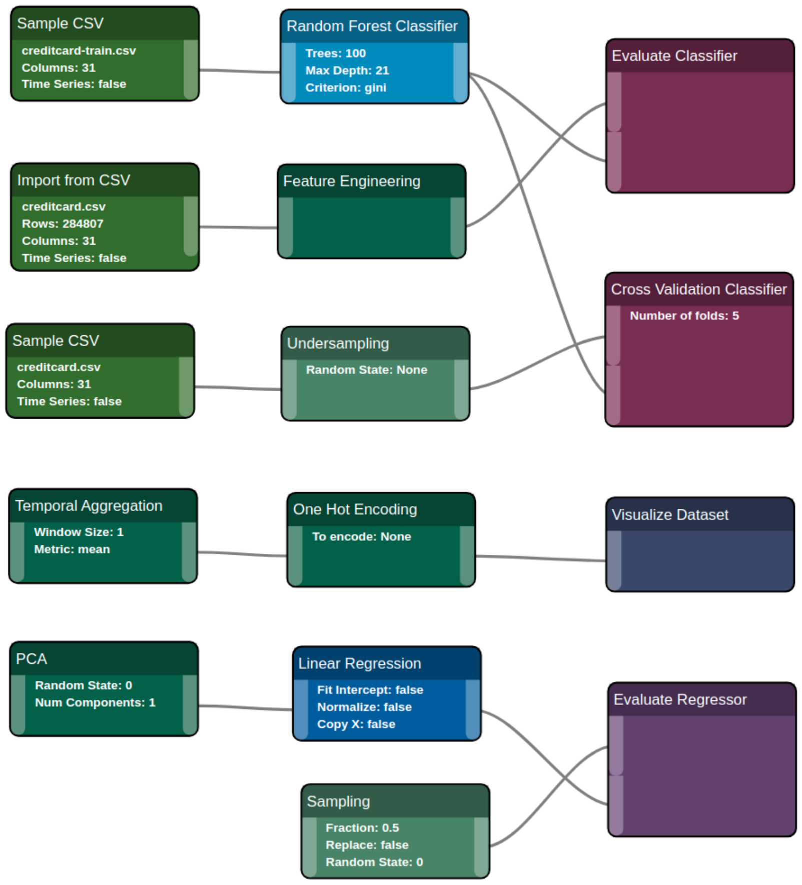


Figure 5.5: Pipeline to be parsed

By parsing the JSON representation of a pipeline, the parser constructs a DAG, so it can be used in the following phases of the compiler. Every block on the pipeline has a one-to-one relation with a node on the graph. The term block refers to the instance within the front-end pipeline and the term node to the instance in the DAG. The pipeline in Figure 5.5 is used throughout this section as an example. Each JSON object within the ***diagram-nodes*** layer leads to an instantiation of the respective back-end node by importing the class with the name stored in the ***type*** property. Every port for that block is instantiated analogously and added to the node instance.

A block is considered **root** when it has no input ports, **loose** when it has and they are not linked to any other block and **connected** when at least one of its input ports and linked to another block (Figure 5.6). Therefore, its node representation in the back-end is considered the same. A **root** block can be seen as the starting point of a complete pipeline because it does not depend on any other block to function. For example, a block for importing a dataset only has an output port to pass the dataset to another block. Whereas a **loose** block is the starting point of an incomplete pipeline because it depends on other blocks to function (those who pass it the data for processing). Figure 5.7 contains an example of what is considered a **root**, **loose** and **connected** blocks.

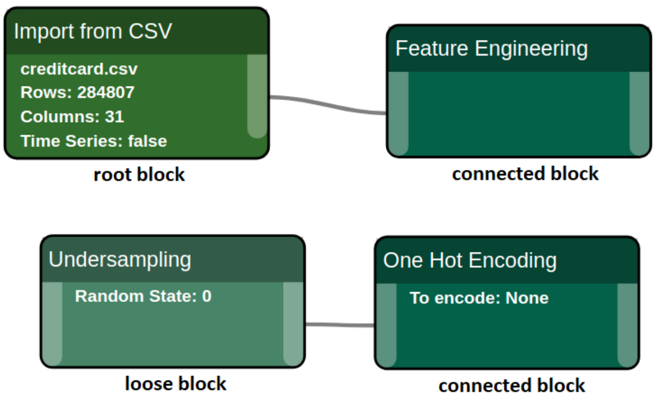


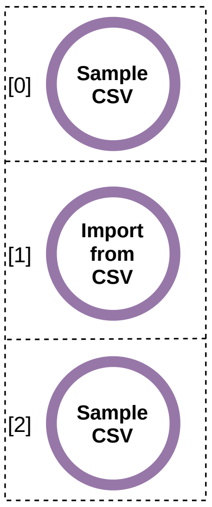
Figure 5.6: Root, loose and connected blocks

After parsing the ***diagram-nodes*** layer from Figure 5.5, the **root** and **loose** nodes are kept in the roots and loose arrays, respectively (Figure 5.7).

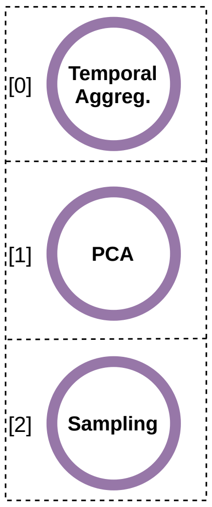
The link dependencies between the blocks on the front-end are maintained by having each node store its children node instances and the parents in SourceLink instances.

The SourceLink instance groups the link ***id***, the source node instance, as well as the source and target port instances. Every block in the pipeline depends on the output port of its predecessor. This SourceLink instance maintains this dependency state on the back-end so that in the following phases of the compiler, it is possible to distinguish from which input port the data comes. For instance, the *Split Dataset* block, which splits the dataset received from the input port into train/test datasets and respectively sends them to the train and test output ports.

After the parsing is complete, both DAGs (Figures 5.8a and 5.8b) representing the pipeline are fully reconstructed on the back-end and accessible through the **roots** and **loose** arrays (the complete and incomplete starting points of a pipeline respectively).

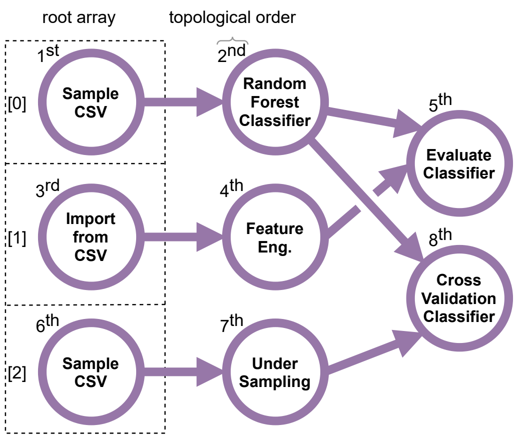


(a) Roots array

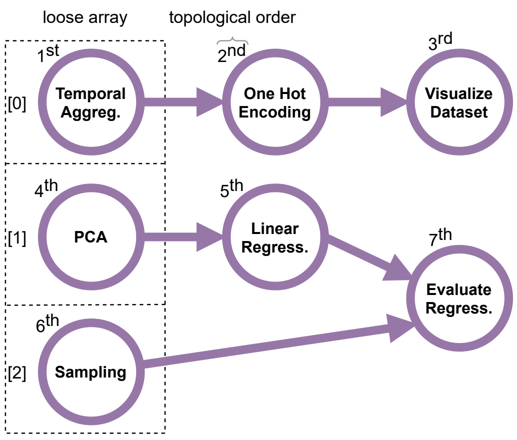


(b) Loose array

Figure 5.7: Root/Loose nodes arrays



(a) DAG from **roots** array



(b) DAG from **loose** array

Figure 5.8: DAGs from **roots** and **loose** arrays

5.3.2 Topological Sorter

A topological sort or topological ordering of a DAG is a linear ordering of its nodes such that for every directed edge uv from node u to node v, u comes before v in the ordering.

Both DAGs reconstructed on the back-end (in the Parser phase) are topological sorted as shown in Figures 5.8a and 5.8b. The result is stored in the **sorted_root** (Figure 5.9) and **sorted_loose** (Figure 5.10) arrays respectively.

Both **sorted_root** and **sorted_loose** arrays are used on the Data Flow Elaboration and Type Checker phases of the compiler. In the Code Generator phase, in which the pipeline is converted into executable code, only the **sorted_root** array is used. Since the **sorted_root** array contains the DAG representation of a complete pipeline, and the **sorted_loose** array an incomplete representation, it only makes sense to generate code for a complete pipeline. Each phase mentioned in this paragraph has its section in this document describing the usage of these arrays in greater detail.

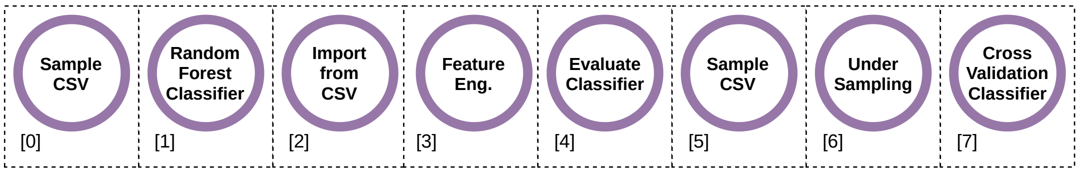


Figure 5.9: **sorted_root** array

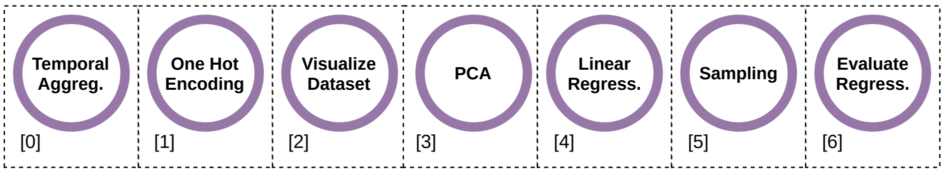


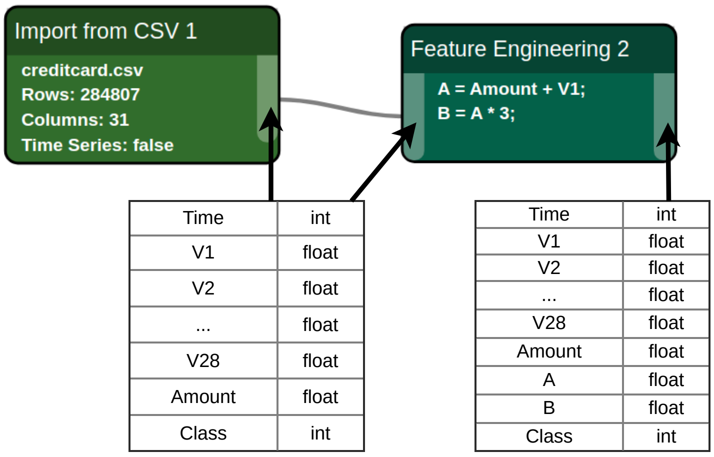
Figure 5.10: **sorted_loose** array

5.3.3 Data Flow Elaboration

Current VP tools for building ML pipelines, such as Orange (Section 3.2), fully load imported datasets into the tool for a dynamical execution of the pipeline, allowing the data scientist to observe the dataset changes in real-time while it is processed through the blocks. The downside of this approach is the coupling of the pipeline creation and execution environments and the problems this tool has when dealing with large datasets due to memory limitations.

MLVP focuses on a static analysis of the pipeline for detecting errors before execution. Instead of having the entire dataset loaded into the tool, only the dataset metadata (column names and types) is loaded. Leaving the user unaware of dataset changes during pipeline creation would turn this process very unintuitive. That is why MLVP uses the dataset metadata to preview what would happen during an eventual execution.

Figure 5.11 exemplifies the passing of dataset metadata through the block's ports. For example, the *Feature Engineering* block uses the dataset metadata entering its input port to state the creation of two new columns using existing ones. The creation of the columns only occurs during the execution of the pipeline.


Import from CSV 1
creditcard.csv
Rows: 284807
Columns: 31
Time Series: false

Feature Engineering 2
A = Amount + V1;
B = A * 3;

Time	int
V1	float
V2	float
...	float
V28	float
Amount	float
Class	int

Time	int
V1	float
V2	float
...	float
V28	float
Amount	float
A	float
B	float
Class	int

Figure 5.11: Metadata data flow between blocks

For achieving the dataset preview, each back-end node mimics the block executing an operation on the dataset received from the input ports by modifying the metadata according to what would happen in a pipeline execution. Since both roots and loose sorted arrays contain the result of topologically sorting the DAG representation of the pipeline, iterating over them to execute the mimic operation guarantees that all nodes have the resulting metadata on their ports to be used in the type-checking phase.

5.3.4 Type Checker

SMT generalizes boolean satisfiability (SAT) by adding equality reasoning, arithmetic, and other applicable first-order theories. An SMT solver is a tool for deciding the satisfiability of formulas in these theories, enabling applications such as extended static checking, predicate abstraction, test case generation, and bounded model checking over infinite domains. Z3 [49] is an SMT solver from Microsoft Research, and it targets the solving of problems that arise in software verification and software analysis.

Each block has an array of formulas that asserts requirements for the metadata received through the input port and ensures characteristics for the metadata exiting from the output port. An example of requirement assertion would be the number of rows on the dataset received from the input port to be greater than a certain value, and one of ensuring would be the dataset sent to the output port being balanced. The assertions are constructed using Z3 variables to refer to both input and output metadata in the following format: $Z3\_type(''port\_id; property'') == property\_value$.

During the Type Checker phase, both roots and loose sorted arrays are iterated to append each block's assertions into an array containing all block assertions. At the same time, all source link assertions of each block are added into a separate array. These assertions establish equality between the dataset metadata properties of an output port with the input port of the following linked blocks.

Figure 5.12 is a pipeline created in MLVP and is used throughout this section to demonstrate how the type checker statically analyses and detects problems with a pipeline. The corresponding arrays containing all the blocks and links assertions for the pipeline are represented in Figure 5.13 respectively.

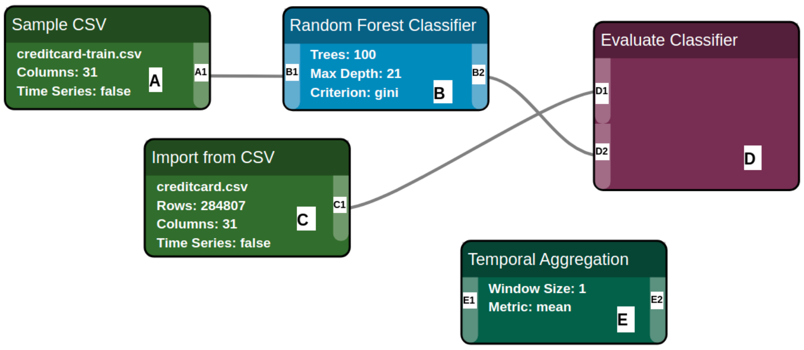
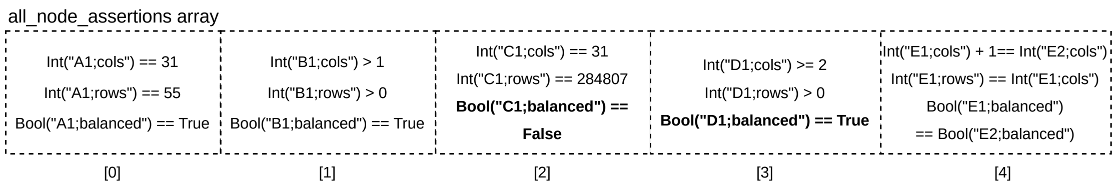


Figure 5.12: Pipeline demonstrating type checking


all_node_assertions array
Int("A1;cols") == 31
Int("A1;rows") == 55
Bool("A1;balanced") == True
Int("B1;cols") > 1
Int("B1;rows") > 0
Bool("B1;balanced") == True
Int("C1;cols") == 31
Int("C1;rows") == 284807
Bool("C1;balanced") == False
Int("D1;cols") >= 2
Int("D1;rows") > 0
Bool("D1;balanced") == True
Int("E1;cols") + 1== Int("E2;cols")
Int("E1;rows") == Int("E1;cols")
Bool("E1;balanced") == Bool("E2;balanced")
[0]
[1]
[2]
[3]
[4]

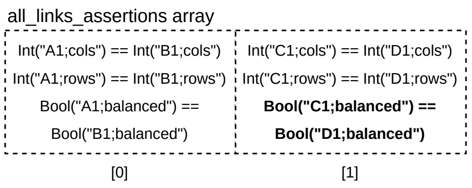

all_links_assertions array
Int("A1;cols") == Int("B1;cols")
Int("A1;rows") == Int("B1;rows")
Bool("A1;balanced") == Bool("B1;balanced")
Int("C1;cols") == Int("D1;cols")
Int("C1;rows") == Int("D1;rows")
Bool("C1;balanced") == Bool("D1;balanced")
[0]
[1]

Figure 5.13: Block and Link assertion arrays

The type checking process is continually being performed as the data scientist manipulates the front-end canvas. The creation of new links or changing block's properties triggers new type checking requests to the back-end. For that reason, the response latency must be kept as low as possible by performing only mandatory computation on the back-end. Therefore, the static analysis of the pipeline is first approached optimistically. All link assertions are added first to the Z3 solver. Only then all block assertions are added. Adding assertions by this order ensures that if the pipeline contains an error, the reported set of assertions causing an unsatisfiable state are block's assertions, which are more helpful for the data scientist. Later in this section, the procedure for finding those assertions is explained in greater detail.

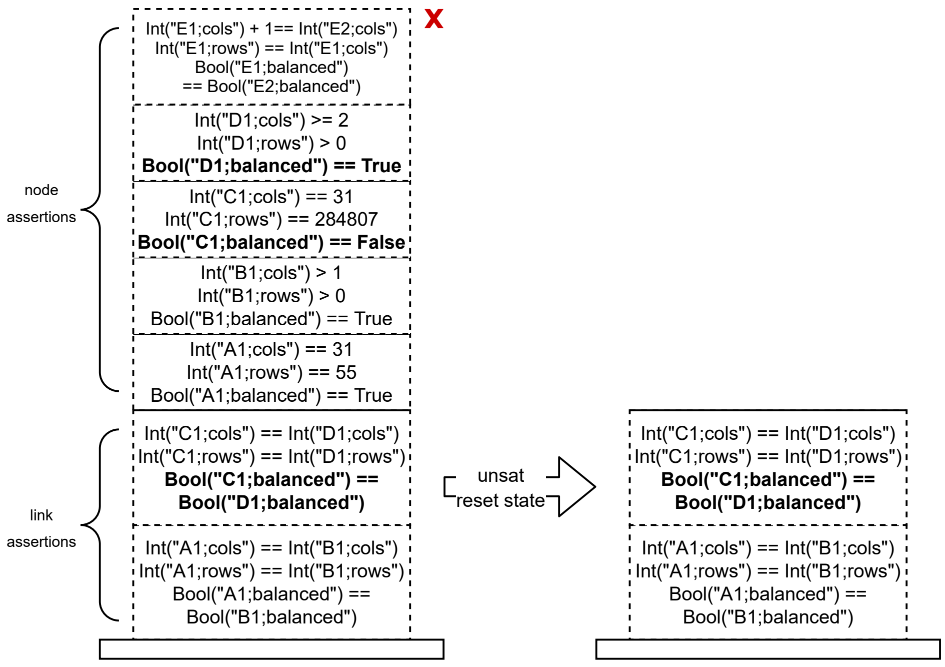

X
Int("E1;cols") + 1== Int("E2;cols")
Int("E1;rows") == Int("E1;cols")
Bool("E1;balanced") == Bool("E2;balanced")
Int("D1;cols") >= 2
Int("D1;rows") > 0
Bool("D1;balanced") == True
Int("C1;cols") == 31
Int("C1;rows") == 284807
Bool("C1;balanced") == False
Int("B1;cols") > 1
Int("B1;rows") > 0
Bool("B1;balanced") == True
Int("A1;cols") == 31
Int("A1;rows") == 55
Bool("A1;balanced") == True
node assertions
Int("C1;cols") == Int("D1;cols")
Int("C1;rows") == Int("D1;rows")
Bool("C1;balanced") == Bool("D1;balanced")
Int("A1;cols") == Int("B1;cols")
Int("A1;rows") == Int("B1;rows")
Bool("A1;balanced") == Bool("B1;balanced")
link assertions
unsat
reset state
Int("C1;cols") == Int("D1;cols")
Int("C1;rows") == Int("D1;rows")
Bool("C1;balanced") == Bool("D1;balanced")
Int("A1;cols") == Int("B1;cols")
Int("A1;rows") == Int("B1;rows")
Bool("A1;balanced") == Bool("B1;balanced")

Figure 5.14: Best case scenario is unsatisfiable

Once both link and block assertions are added to the solver, its satisfiability is checked. If it returns **sat**, the type-checking phase ends with no assertions being violated (optimistic). If instead **unsat** is returned, there are contradicting assertions within the solver. To pinpoint what are the violated assertions, the solver must be reset to a previous satisfiable state by popping the scope containing all block assertions, leaving only the link assertions (Figure 5.14). Then an incremental solving phase starts by iterating over the array containing all block assertions and individually adding and checking for the solver's satisfiability.

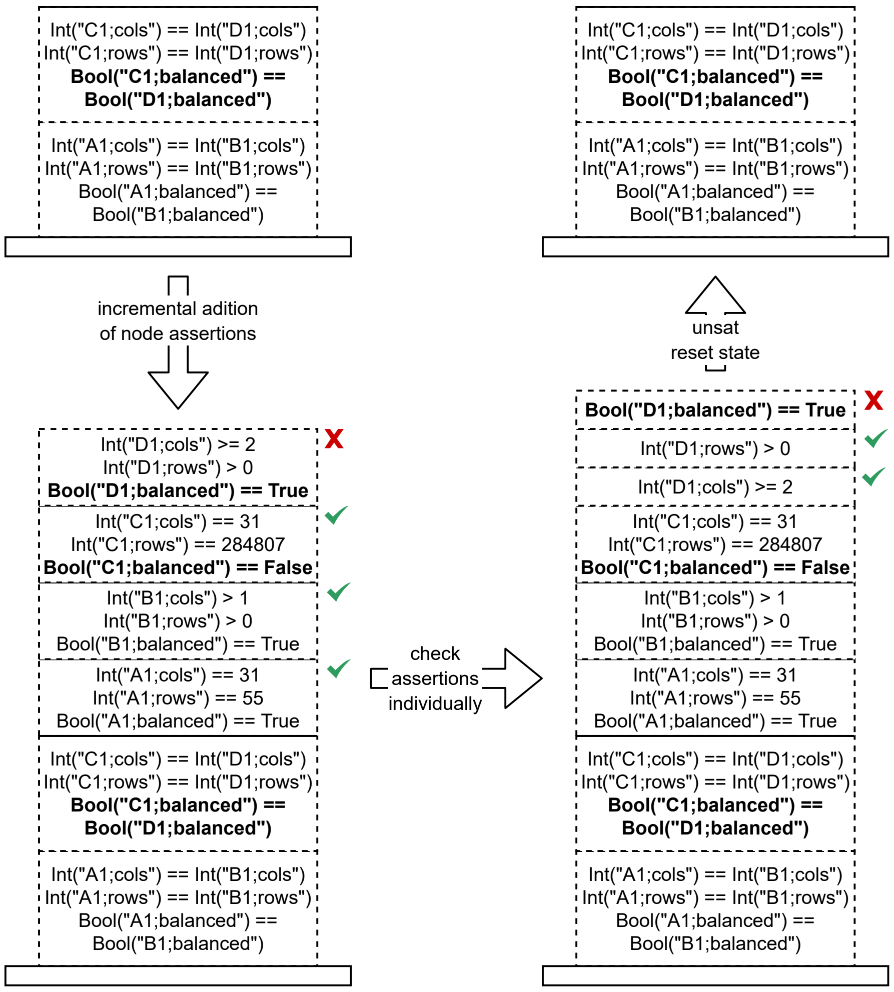
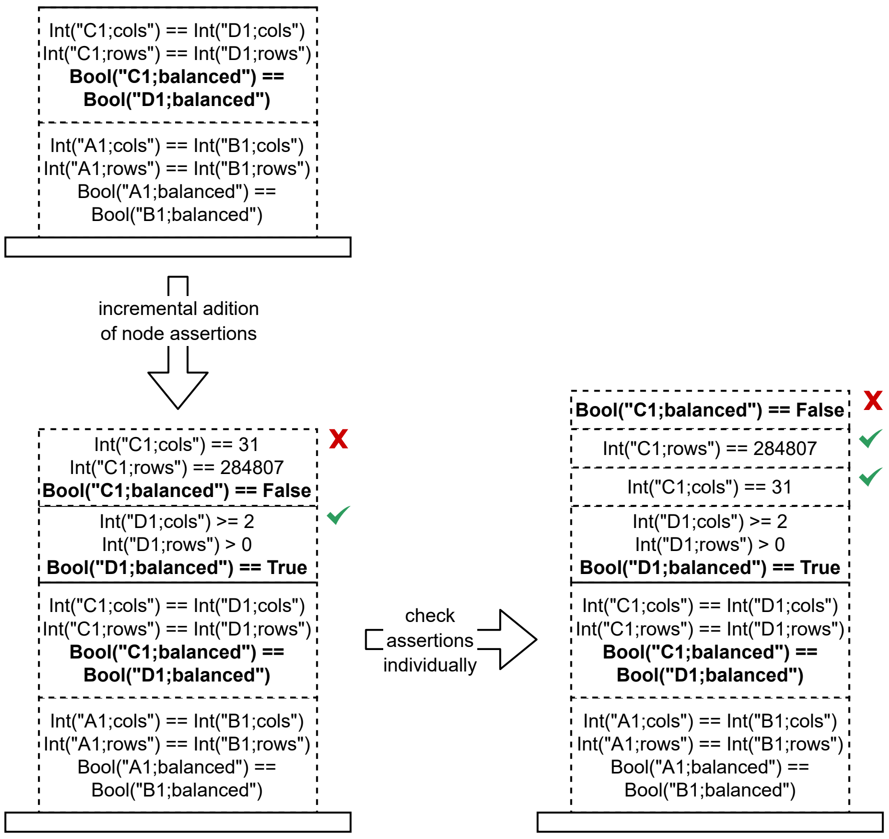

Int("C1;cols") == Int("D1;cols")
Int("C1;rows") == Int("D1;rows")
Bool("C1;balanced") == Bool("D1;balanced")
Int("A1;cols") == Int("B1;cols")
Int("A1;rows") == Int("B1;rows")
Bool("A1;balanced") == Bool("B1;balanced")
incremental adition of node assertions
unsat reset state
Int("D1;cols") >= 2
Int("D1;rows") > 0
Bool("D1;balanced") == True
Int("C1;cols") == 31
Int("C1;rows") == 284807
Bool("C1;balanced") == False
Int("B1;cols") > 1
Int("B1;rows") > 0
Bool("B1;balanced") == True
Int("A1;cols") == 31
Int("A1;rows") == 55
Bool("A1;balanced") == True
check assertions individually

Figure 5.15: Finding affected block

Figure 5.15 demonstrates the process described next. At every iteration, a new scope is pushed into the solver's stack, and the current block assertions are added to the solver, followed by a satisfiability check. Eventually, the **unsat** will reappear, and the block whose assertions originate the unsatisfiable state is found. The next step is to identify the specific assertion/s causing the **unsat** within that block's array of assertions. The procedure is similar to the one just described, the current block's assertions are removed by popping the solver scope, and then each assertion is added to the solver individually after having a new scope pushed, with a posterior satisfiability check.



Int("C1;cols") == Int("D1;cols")
Int("C1;rows") == Int("D1;rows")
Bool("C1;balanced") == Bool("D1;balanced")
Int("A1;cols") == Int("B1;cols")
Int("A1;rows") == Int("B1;rows")
Bool("A1;balanced") == Bool("B1;balanced")
incremental adition of node assertions
Int("C1;cols") == 31
Int("C1;rows") == 284807
Bool("C1;balanced") == False
X
Int("D1;cols") >= 2
Int("D1;rows") > 0
Bool("D1;balanced") == True
✓
check assertions individually
Bool("C1;balanced") == False
X
Int("C1;rows") == 284807
✓
Int("C1;cols") == 31
✓

Figure 5.16: Finding originator block

Once all individual assertions are found, they are stored to be later sent to the front-end. However, the type checking is not over yet. There is still the need to find the other block whose assertions are conflicting with the block just found (**affected** block) to give as much information as possible to the data scientist about the problems of the pipeline. If the type-checking phase were to end at this point, the data scientist would only know about the block, and its assertions, affected by the problem but would not know what the block causing it was.

The whole procedure described for finding the **affected** block is applied to the reverse of the array containing all block assertions, shown in Figure 5.16. The solver's state is reset to a previous satisfiable state containing only the link assertions. Then, the reverse of the array ranging from the index of the **affected** block to the beginning of the array (index 0) is iterated, adding each block's assertions to the solver and checking for its satisfiability on every iteration. When the **unsat** reappears, the block causing the unsatisfiable state (**originator** block) is found.

The iteration of the array in reverse consists of traversing the DAG backwards. That is, from the **affected** block to the **originator** block. Once the conflicting assertions of the **originator** block are found, they are sent to the front-end along with the conflicting **affected** block's assertions.

Pre-compilation type checking

Recall that a block is considered **root** when it has no input ports, **loose** when it has, and they are not linked to any other block and **connected** when at least one of its input ports and linked to another block. The back-end node representation follows the same terminology.

Before moving to the next and final phase of the compiler, it is essential to clarify a quirk about the way the type checker functions. Depending on whether there is a code generation phase immediately after the type checking, the type checker is more or less constrained on the static analysis: If there is not a code generation phase immediately after the type-checking, then the type checking is handled exactly the way it was described to this point. However, if there is, then additional assertions are included to each block's assertions array to ensure that all input ports are linked to some other block, and all **loose** blocks and subsequent **connected** block which do not have any link path to a **root** block are excluded from the type checking process. Since their sub-graph representation is missing source links dependencies to a **root** block (the source of data in the pipeline), its compilation would result in dead code. For the pipeline from Figure 5.12, the assertions of the *Temporal Aggregation* block would not be part of the array containing all block assertions.

Figure 5.17 contains a scenario in which the pipeline would pass the permissive type-checking but fail the strong type checking due to the *Evaluate Classifier* block having an input port missing a link.

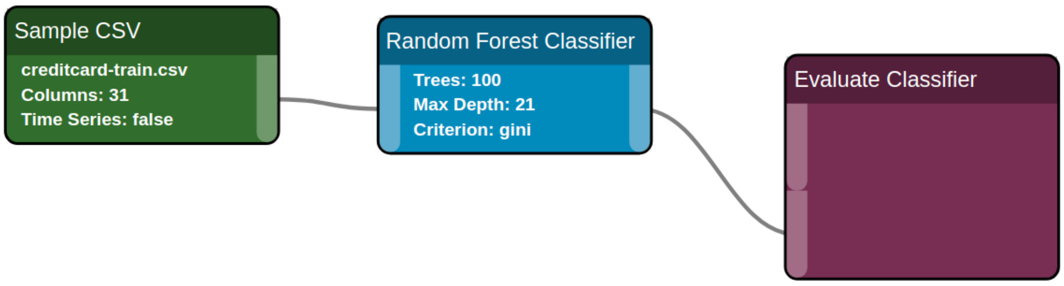


Figure 5.17: Pipeline failing pre-compilation type checking

5.3.5 Code Generator

Code-based approaches to pipeline creation require the programmer to handle library imports, object instantiation, and variable management manually. In the MLVP tool, the code is generated automatically according to the pipeline created. Each block type has a code template shared by all its instances in the compilation of the pipeline. During compilation, the code generator writes the required library imports and object instantiations for each block composing the pipeline by fulfilling their template placeholders with the values set by the data scientists in the front-end.

Figure 5.18 represents a pipeline being compiled. The array resulted from topological sorting its DAG representation is on the left of Figure 5.19, while the corresponding pipeline executable source code is on the right. The array is iterated twice. Firstly to write each block's import code template so that all imports stay at the top of the output file. Secondly to write the block's logic code template in the output executable source code so that the topological sorting of the blocks is maintained in the output file. To facilitate the observation of that order, the graph nodes in the array share the colour of the code template in the output source code.

Taking as example the link between the *Sample CSV* and *Random Forest Classifier* blocks from Figure 5.18 and the corresponding compiled code from Figure 5.19. The former reads the "creditcard-train" CSV file and splits it into x1 and y1 data frames, which are then used by the latter to train the classifier. The usage of an emitter to bind the output ports of the blocks with the corresponding variable names containing the result of their operations makes this automatic variable handling possible.

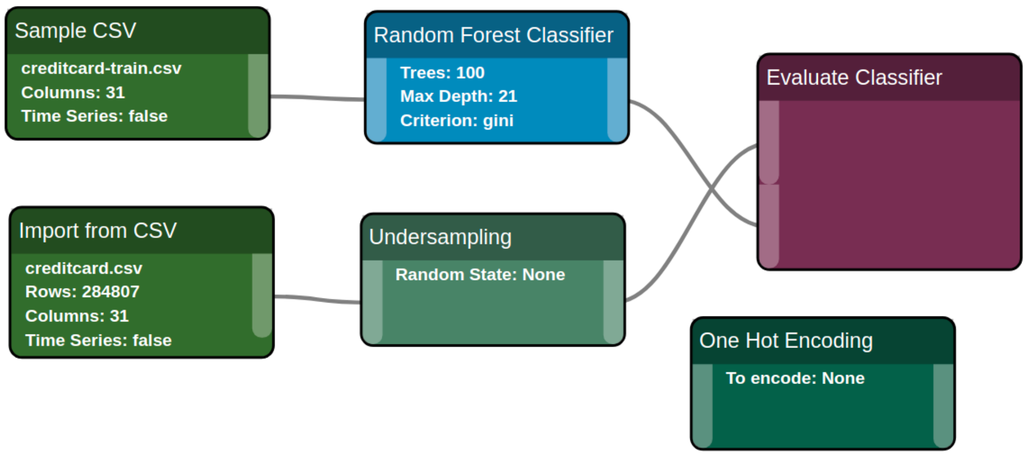
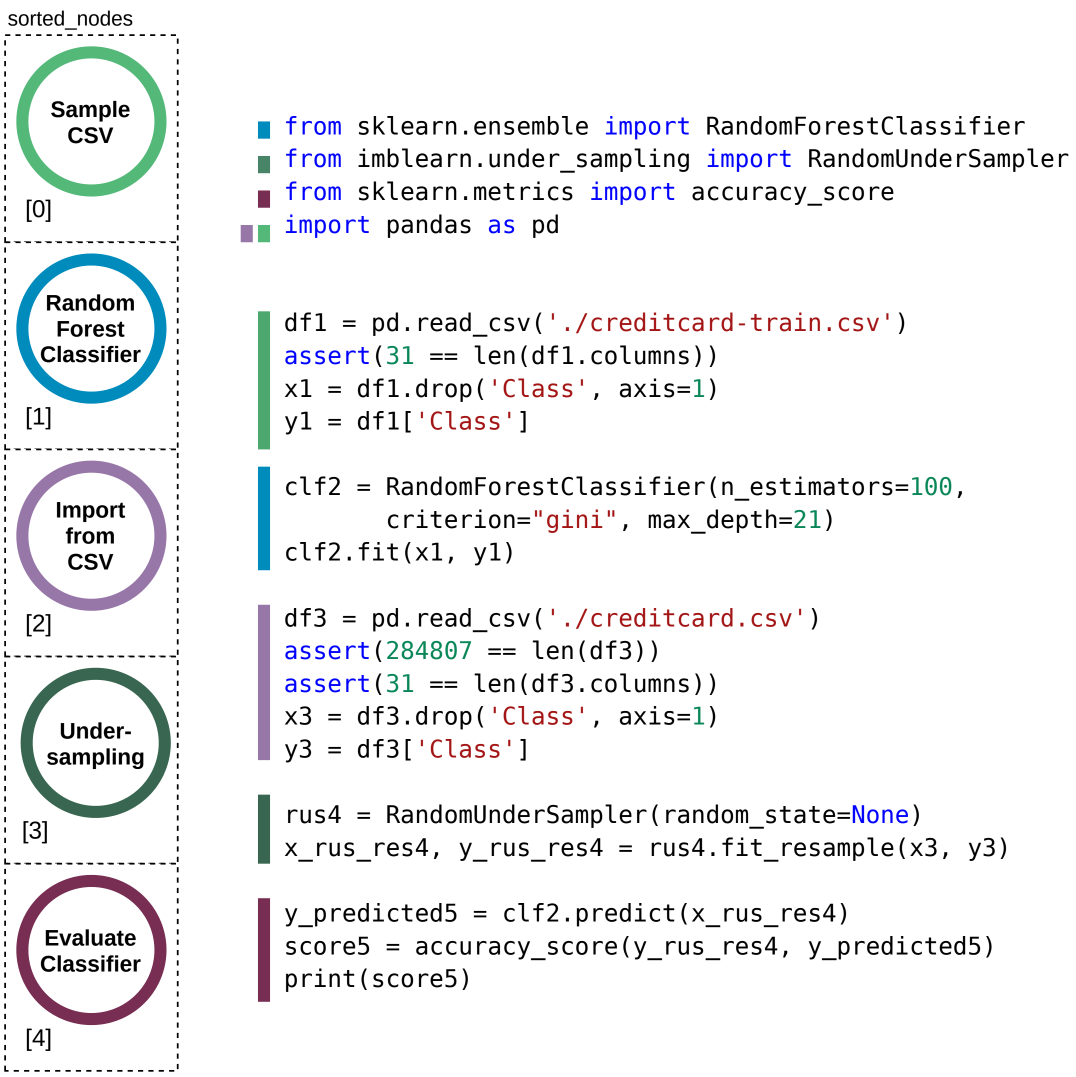


Figure 5.18: Compiled pipeline



```
from sklearn.ensemble import RandomForestClassifier
from imblearn.under_sampling import RandomUnderSampler
from sklearn.metrics import accuracy_score
import pandas as pd


df1 = pd.read_csv('./creditcard-train.csv')
assert(31 == len(df1.columns))
x1 = df1.drop('Class', axis=1)
y1 = df1['Class']

clf2 = RandomForestClassifier(n_estimators=100,
       criterion="gini", max_depth=21)
clf2.fit(x1, y1)

df3 = pd.read_csv('./creditcard.csv')
assert(284807 == len(df3))
assert(31 == len(df3.columns))
x3 = df3.drop('Class', axis=1)
y3 = df3['Class']

rus4 = RandomUnderSampler(random_state=None)
x_rus_res4, y_rus_res4 = rus4.fit_resample(x3, y3)

y_predicted5 = clf2.predict(x_rus_res4)
score5 = accuracy_score(y_rus_res4, y_predicted5)
print(score5)
```

Figure 5.19: Compiled pipeline's executable source code

5.4 Blocks Library

This chapter presents the implemented blocks for MLVP and their respective assertions. The chapter is organized into two sections. The first contains the simpler blocks, whose implementation process mainly consists of replacing placeholders for the assertions and code templates. While the second contains the blocks whose node representation on the back-end goes through a more unique a complex process.

5.4.1 Simple Blocks

The blocks presented in this sub-section follow a common rule of implementation. The values written by the data scientist in the input fields of the block's properties on the UI are stored in the pipeline's JSON save representation. In the back-end, the JSON is parsed to recreate the pipeline DAG representation by instantiating each graph node with those values. Finally, during both type checking and code generation phases, the values are inserted in the placeholders of the assertions and code templates.

Abstract Dataset

The *Abstract Dataset* block allows for the definition of an abstract dataset by choosing the number of rows and columns, as well as indicating whether the dataset represents a time series or is balanced without having to load any dataset file. Since these properties are not hardcoded in the block type implementation, they must be mutable according to each block instance. Listing 5.4 contains the block's assertions with $?< property >$ representing the placeholders of the block.

```
Output Dataset port
   cols == ?cols
   rows == ?rows
   time_series == ?time_series
   balanced == ?balanced
   reduced == ?reduced
   processed == ?processed
   unique_column_names == True
```

Code Listing 5.4: Abstract Datset block assertions

Import from CSV

The *Import from CSV* block allows the definition of a dataset by parsing a dataset file. The block then stores its metadata, which includes the number of rows and columns, each column name and respective type, the number of instances for each class, and whether it is balanced or not. Similarly to the *Abstract Dataset* block, it is possible to indicate if it represents a time series. Since these properties are not hardcoded in the block type implementation, they must be mutable according to each block instance. Listing 5.5 contains the block's assertions with $?< property >$ representing the placeholders of the block.

```
Output Dataset port
   cols == ?cols
   rows == ?rows
   time_series == ?time_series
   balanced == ?balanced
   reduced == ?reduced
   processed == ?processed
   unique_column_names == True
   col[1] == ?type
   col[2] == ?type
   ...
   col[n] == ?type
```

Code Listing 5.5: Import from CSV block assertions

Sample CSV

The *Sample CSV* block is very similar to the *Import from CSV* block. The main difference being the information about rows is ignored in the parsing of the dataset file. Therefore, the number of rows is not stored in the block's properties. Still, the block allows for manual identification of whether the dataset represents a time series or is balanced. Listing 5.6 contains the block's assertions with ?$<$ *property* $>$ representing the placeholders of the block.

```
Output Dataset port
   cols == ?cols
   time_series == ?time_series
   balanced == ?balanced
   reduced == ?reduced
   processed == ?processed
   unique_column_names == True
   col[1] == ?type
   col[2] == ?type
   ...
   col[n] == ?type
```

Code Listing 5.6: Sample CSV block assertions

PCA

The *PCA* block applies a PCA to the dataset received through the input port and forwards the result to the output port. The block allows using a seed value for operation reproducibility and choosing the number of column components to create. Listing 5.7 contains the block's assertions.

```
Inner Properties
   num_components > 0

Input Dataset port (input_ds)
   num_components < cols
   columns_of_type_string == False

Output Dataset port
   cols == num_components + 1
   rows == input_ds.rows
   balanced == input_ds.balanced
   time_series == input_ds.time_series
   reduced == input_ds.reduced
   processed == True
   col[1] == float
   col[2] == float
   ...
   col[num_components] == float
```

Code Listing 5.7: PCA block assertions

Temporal Aggregation

The *Temporal Aggregation* block applies a temporal aggregation operation to the dataset received through the input port. The data scientist chooses the size of the rolling window, the column from the input dataset used for the operation, the metric applied to the values, and the name of the new column resulted from the operation. Listing 5.7 contains the block's assertions.

```
Inner Properties
   len(new_col_name) > 0
   new_col_name_unique == True
   type(used_col) == float or type(used_col) == int

Input Dataset port (input_ds)
   time_series == True

Output Dataset port
   cols == input_ds.cols + 1
   rows == input_ds.rows
   balanced == input_ds.balanced
   time_series == input_ds.time_series
   reduced == input_ds.reduced
   processed == True
   col(new_col_name) == float
```

Code Listing 5.8: Temporal Aggregation block assertions

Label Encoding

The *Label Encoding* block applies a label encoding on a chosen column of the dataset received on the input port. That column must be of type categorical (int or string). Listing 5.9 contains the block's assertions.

```
Input Dataset port (input_ds)
   type_col(chosen_column) == int or type_col(chosen_column) ==
      string

Output Dataset port
   cols == input_ds.cols
   rows == input_ds.rows
   balanced == input_ds.balanced
   time_series == input_ds.time_series
   reduced == input_ds.reduced
   processed == True
   type_col(encoded_column) == int
```

Code Listing 5.9: Label Encoding block assertions

Label Decoding

The *Label Decoding* block reverts the label encoding performed by the *Label Encoding* block. Therefore, the dataset received on the input port of the *Label Decoding* block must have at least one column that is label encoded. Listing 5.10 contains the block's assertions.

```
Input Dataset port (input_ds)
   num_encoded_cols >= 1

Output Dataset port
   cols == input_ds.cols
   rows == input_ds.rows
   balanced == input_ds.balanced
   time_series == input_ds.time_series
   reduced == input_ds.reduced
   processed == True
   type_col(decoded_column) == type_col(column_before_encoding)
```

Code Listing 5.10: Label Decoding block assertions

One Hot Encoding

The *One Hot Encoding* block applies a one-hot encoding on a chosen column of the dataset received on the input port. That column must be of type categorical (int or string). Listing 5.9 contains the block's assertions.

```
Input Dataset port (input_ds)
   type_col(chosen_column) == int or type_col(chosen_column) ==
      string

Output Dataset port
   cols == input_ds.cols
   rows == input_ds.rows
   balanced == input_ds.balanced
   time_series == input_ds.time_series
   reduced == input_ds.reduced
   processed == True
   type_col(encoded_column) == one_hot_encoded
```

Code Listing 5.11: One Hot Encoding block assertions

One Hot Decoding

The *One Hot Decoding* block reverts the one-hot encoding performed by the *One Hot Encoding* block. Therefore, the dataset received on the input port of the *One Hot Decoding* block must have at least one column that is one-hot encoded. Listing 5.12 contains the block's assertions.

```
Input Dataset port (input_ds)
   num_encoded_cols >= 1

Output Dataset port
   cols == input_ds.cols
   rows == input_ds.rows
   balanced == input_ds.balanced
   time_series == input_ds.time_series
   reduced == input_ds.reduced
   processed == True
   type_col(decoded_column) == type_col(column_before_encoding)
```

Code Listing 5.12: One Hot Decoding block assertions

Vertical Concatenation

The *Vertical Concatenation* block vertically concatenates the datasets received from the two input ports. The dataset rows received from the first input port stay on top of the rows of the dataset received on the second input port. For the operation to be valid, both datasets must have an equal number of columns, and the columns from both datasets that are at the same position must be of the same type. Listing 5.13 contains the block's assertions.

```
Inner Properties
   top_input_ds.cols == bot_input_ds.cols
   col_types(top_input_ds) == col_types(bot_input_ds)

Input Top Dataset port (top_input_ds)
   ---

Input Bottom Dataset port (bot_input_ds)
   ---

Output Dataset port
   cols == top_input_ds.cols
   cols == bot_input_ds.cols
   rows == top_input_ds.rows + bot_input_ds.rows
   balanced == False
   time_series == False
   reduced == False
   processed == True
```

Code Listing 5.13: Vertical Concatenation block assertions

Horizontal Concatenation

The *Horizontal Concatenation* block horizontally concatenates the datasets received from the two input ports. The dataset columns received from the first input port stay on the left, while the dataset columns received from the second input port stay on the right. For the operation to be valid, both datasets must not share column names and have the same number of rows. Listing 5.14 contains the block's assertions.

```
Inner Properties
   left_input_ds.rows == right_input_ds.rows

Input Left Dataset port (left_input_ds)
   ---

Input Right Dataset port (right_input_ds)
   ---

Output Dataset port
   cols == left_input_ds.cols + right_input_ds.cols
   rows = left_input_ds.rows
   rows = right_input_ds.rows
   balanced = right_input_ds.balanced
   time_series == (left_input_ds.time_series and right_input_ds.
      time_series)
   reduced == False
   processed == True
```

Code Listing 5.14: Horizontal Concatenation block assertions

Oversampling

The *Oversampling* block balances the dataset received from the input port by duplicating rows in the minority classes. The resulting dataset is sent to the output port, ensured to be balanced. Listing 5.15 contains the block's assertions.

```
Input Dataset port (input_ds)
   ---

Output Dataset port
   cols == input_ds.cols
   input_ds.balanced == True -> rows == input_ds.rows
   input_ds.balanced == False -> rows == input_ds.
      max_label_count * input_ds.n_labels
   balanced == True
   time_series == False
   reduced == input_ds.reduced
   processed == True
```

Code Listing 5.15: Oversampling block assertions

Undersampling

The *Undersampling* block balances the dataset received from the input port by deleting rows in the majority classes. The resulting dataset is sent to the output port, ensured to be balanced. Listing 5.16 contains the block's assertions.

```
Input Dataset port (input_ds)
   processed == False

Output Dataset port
   cols == input_ds.cols
   input_ds.balanced == True -> rows == input_ds.rows
   input_ds.balanced == False -> rows == input_ds.
      min_label_count * input_ds.n_labels
   balanced == True
   time_series == input_ds.time_series
   reduced == True
   processed == input_ds.processed
```

Code Listing 5.16: Undersampling block assertions

Split Dataset

The *Split Dataset* block splits the dataset received from the input port in train and test datasets, which is sent to each output port. The first output port contains the training dataset, while the second output port the test dataset. Listing 5.17 contains the block's assertions.

```
Input Dataset port (input_ds)
   rows >= 2

Output Train Dataset port
   rows == input_ds.rows * train_size
   cols == input_ds.cols
   reduced == input_ds.reduced
   processed == True
   stratify == True -> balanced == True

Output Test Dataset port
   rows == input_ds.rows * test_size
   cols == input_ds.cols
   reduced == input_ds.reduced
   processed == True
   stratify == True -> balanced == True
```

Code Listing 5.17: Split Dataset block assertions

Sampling

The *Sampling* block samples the dataset received from the input port using the fraction defined in the block's properties, allowing for the operation to be done with or without replacement, as well as allowing the usage of a seed value for consistent replication of the operation. Listing 5.18 contains the block's assertions.

```
Input Dataset port (input_ds)
   processed == False

Output Dataset port
   cols == input_ds.cols
   rows == input_ds.rows * frac
   balanced == False
   time_series == False
   reduced == True
   processed == input_ds.processed
```

Code Listing 5.18: Sampling block assertions

Visualize Dataset

The *Visualize Dataset* block allows for the visualization of the dataset received on the input port. This block does not have any defined assertions.

Abstract Classifier

All classifier blocks (*Decision Tree*, *Random Forest*, *SVM*, *K Nearest Neighbors* and *Keras*) train using the dataset received from the input port, each one having their own specific properties. The trained classifier is then sent to the output port. The base assertions for the classifier blocks are shown in Listing 5.19.

```
Input Dataset port (input_ds)
   balanced == True
   cols > 1
   rows > 0
   all_features_types != string
   col[:-1] == int or col[:-1] == string
```

Code Listing 5.19: Classifier block assertions

Abstract Regressor

All regressor blocks (*Random Forest*, *Linear*, *Logistic*, *SVM* and *Keras*) train using the dataset received from the input port, each one having their own specific properties. The trained regressor is then sent to the output port. The base assertions for the regressor blocks are shown in Listing 5.20.

```
Input Dataset port (input_ds)
   cols > 1
   rows > 0
   all_features_types != string
   col[:-1] == int or col[:-1] == float
```

Code Listing 5.20: Regressor block assertions

Evaluate Classifier

The *Evaluate Classifier* block evaluates the prediction accuracy of the trained classifier received from the second input port using the dataset received from the first input port. Listing 5.21 contains the block's assertions.

```
Input Dataset port
   balanced == True
   cols >= 2
   rows > 0
   all_features_types != string
   col[:-1] == int or col[:-1] == string
```

Code Listing 5.21: Evaluate Classifier block assertions

Cross Validation Classifier

The *Cross Validation Classifier* block evaluates the prediction accuracy of the trained classifier received from the second input port using the dataset received from the first input port through cross-validation. Listing 5.22 contains the block's assertions.

```
Inner properties
   num_folds > 1

Input Dataset port
   cols >= 2
   rows > 0
   balanced == True
   time_series == False
   all_features_types != string
   col[:-1] == int or col[:-1] == string
```

Code Listing 5.22: Cross Validation Classifier block assertions

Evaluate Regressor

The *Evalaute Regressor* block evaluates the prediction accuracy of the trained regressor received from the second input port using the dataset received from the first input port. Listing 5.23 contains the block's assertions.

```
Input Dataset port
   cols >= 2
   rows > 0
   all_features_types != string
   col[:-1] == int or col[:-1] == float
```

Code Listing 5.23: Evaluate Regressor block assertions

5.4.2 Complex Blocks

Similar to the Simple Blocks (subsection 5.4.1), Complex Blocks also make use of the pipeline's JSON save representation to send information to the back-end. However, the block's back-end node representation procedure during both type checking and code generation phases goes beyond replacing placeholders in the assertions and code templates. Instead, they have unique procedures, which are explained in greater detail in the following paragraphs.

Feature Engineering

The *Feature Engineering* block allows for the engineering of new dataset columns using existing ones. The block contains a coding area, which uses the DSL detailed in Figure 5.20 to create new columns. The language includes the basic types bool, int, float, and string. Contains expressions for literal values (l) or variables (x), which are the name of existing dataset columns.

Figure 5.21 contains the rules for the type checking of expressions is the DSL and Table 5.2 contains the operators represented by the $\bigoplus$, $\bigodot$ and $\bigtriangledown$ symbols.

Symbol	Operators
$\bigoplus$	$/, -, +, \times, \%, \geq, >, \leq, <$
$\bigodot$	and, or
$\bigtriangledown$	$+, \geq, >, \leq, <$

Table 5.2: Feature Engineering Type Checking symbols

ANTLR [50, 51] is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It is widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees.

For the construction of an Abstract Syntax Tree (AST) representing the code written in the *Feature Engineering* block coding area, the DSL has its grammar defined in ANTLR. The parser tree generated by ANTLR is then visited to build the AST.

$$
\begin{array}{lrcl}
\text{Types} & T & ::= & bool \mid int \mid float \mid string \\
\text{Expressions} & e & ::= & x \\
 & & & \mid l \\
 & & & \mid e + e \\
 & & & \mid e - e \\
 & & & \mid e * e \\
 & & & \mid e / e \\
 & & & \mid e \,\%\, e \\
 & & & \mid e \; and \; e \\
 & & & \mid e \; or \; e \\
 & & & \mid e == e \\
 & & & \mid e \,! = e \\
 & & & \mid e \geq e \\
 & & & \mid e > e \\
 & & & \mid e \leq e \\
 & & & \mid e < e \\
 & & & \mid not \; e \\
 & & & \mid -e \\
\text{Contexts} & \Gamma & ::= & \varepsilon \mid \Gamma, x : T
\end{array}
$$

Figure 5.20: The syntax of Feature Engineering DSL

The initial context state during the type checking of the AST consists of the column names and respective column types for the dataset received on the input port. During type checking, expressions are checked to ensure they follow the rules from Figure 5.21. Every new column created must have a unique name. If it has, its name and type are added to the context to be used on the following column creation statements. Listing 5.24 contains the block assertions for the *Feature Engineering* block.

During the code generation phase of the pipeline, the AST is visited once again to have the expressions translated into the output executable code.

$$\frac{}{\Gamma \vdash b\colon \mathit{bool}} \qquad \frac{}{\Gamma \vdash i\colon \mathit{int}} \qquad \frac{}{\Gamma \vdash f\colon \mathit{float}} \qquad \frac{}{\Gamma \vdash s\colon \mathit{string}} \qquad \text{(literal)}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x\colon T} \qquad \text{(variable)}$$

$$\frac{\Gamma \vdash e_1\colon \mathit{bool} \quad \Gamma \vdash e_2\colon \mathit{bool}}{\Gamma \vdash e_1 \bigodot e_2\colon \mathit{bool}} \qquad \frac{\Gamma \vdash e\colon \mathit{bool}}{\Gamma \vdash \mathit{not}\ e\colon \mathit{bool}} \qquad \text{(bool-operators)}$$

$$\frac{\Gamma \vdash e_1\colon \mathit{int} \quad \Gamma \vdash e_2\colon \mathit{int}}{\Gamma \vdash e_1 \bigoplus e_2\colon \mathit{int}} \qquad \frac{\Gamma \vdash e_1\colon \mathit{float} \quad \Gamma \vdash e_2\colon \mathit{float}}{\Gamma \vdash e_1 \bigoplus e_2\colon \mathit{float}} \qquad \text{(same-type-operators)}$$

$$\frac{\Gamma \vdash e_1\colon \mathit{int} \quad \Gamma \vdash e_2\colon \mathit{float}}{\Gamma \vdash e_1 \bigoplus e_2\colon \mathit{float}} \qquad \frac{\Gamma \vdash e_1\colon \mathit{float} \quad \Gamma \vdash e_2\colon \mathit{int}}{\Gamma \vdash e_1 \bigoplus e_2\colon \mathit{float}} \qquad \text{(diff-type-operators)}$$

$$\frac{\Gamma \vdash e\colon \mathit{int}}{\Gamma \vdash -e\colon \mathit{int}} \qquad \frac{\Gamma \vdash e\colon \mathit{float}}{\Gamma \vdash -e\colon \mathit{float}} \qquad \text{(negative)}$$

$$\frac{\Gamma \vdash e_1\colon \mathit{string} \quad \Gamma \vdash e_2\colon \mathit{string}}{\Gamma \vdash e_1 \bigtriangledown e_2\colon \mathit{string}} \qquad \text{(string-concat-and-comparison)}$$

$$\frac{\Gamma \vdash e_1\colon \mathit{string} \quad \Gamma \vdash e_2\colon \mathit{int}}{\Gamma \vdash e_1 \times e_2\colon \mathit{string}} \qquad \frac{\Gamma \vdash e_1\colon \mathit{int} \quad \Gamma \vdash e_2\colon \mathit{string}}{\Gamma \vdash e_1 \times e_2\colon \mathit{string}} \qquad \text{(string-multiply)}$$

$$\frac{}{\Gamma \vdash e_1 == e_2\colon \mathit{bool}} \qquad \frac{}{\Gamma \vdash e_1\ != e_2\colon \mathit{bool}} \qquad \text{(type-equality)}$$

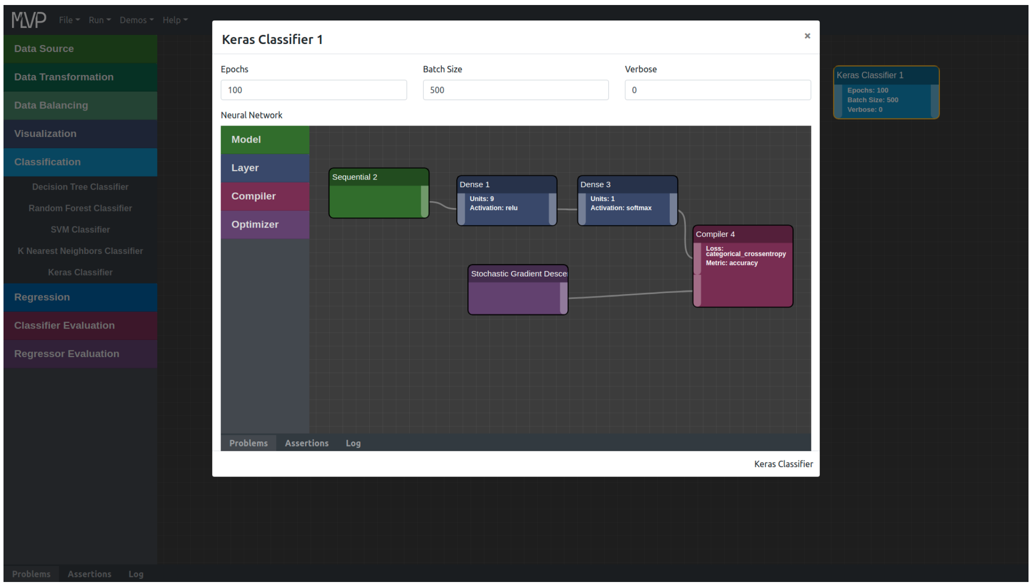
Figure 5.21: Feature Engineering Type Checking

```
Inner properties
   dsl_program_correct == True

Input Dataset port (input_ds)
   ---

Output Dataset port
   cols == input_ds.cols + num_new_features_created
   rows == input_ds.rows
   balanced == input_ds.balanced
   time_series == time_series.balanced
   reduced == input_ds.balanced
   processed == True
```

Code Listing 5.24: Feature Engineering block assertions

Keras Classifier/Regressor

Both *Keras Classifier* and *Keras Regressor* blocks allow for the creation of neural networks using a dedicated neural network canvas within the block's modal (Figure 5.22). The neural network canvas works similarly to the pipeline canvas with its Model, Layer, Compiler, and Optimizer blocks. When the pipeline is serialized into the JSON save representation (Subsection 5.2.4), the neural network canvas of each *Keras* block is serialized as well and stored alongside the other key/value pairs of the block (epochs, batch size and verbose).



Epochs: 100
Batch Size: 500
Verbose: 0

Dense 1 — Units: 9, Activation: relu
Dense 3 — Units: 1, Activation: softmax
Compiler 4 — Loss: categorical_crossentropy, Metric: accuracy

Keras Classifier 1 — Epochs: 100, Batch Size: 500, Verbose: 0

Figure 5.22: Keras dedicated neural network canvas

When the *Keras* back-end node is initialized during the parsing phase of the pipeline's JSON representation. The neural network canvas inside of the block's JSON representation goes through its parser, topological sorter, data flow elaboration, and type checking phases, where the blocks and links composing the neural network are verified. The code generation phase of the neural network only occurs when triggered by the main pipeline code generation phase.

The *Keras Classifier* and *Keras Regressor* blocks share the block assertions of all classifier and regressor blocks, respectively. Additionally, *Keras* blocks have an assertion to ensure that the neural network is well built. The list of blocks implemented for the neural network are listed below, with the *Compiler* block being the only one with assertions defined.

Sequential The *Sequential* block is the starting point of the neural network layers. It has an output Layer port to initiate the neural network's layer by connecting to a *Dense* block.

Dense The *Dense* block represents a layer in the neural network. By sequentially linking *Dense* blocks, new layers are added to the neural network. The last *Dense* block is then linked to the *Compiler* block to finish the network.

Stochastic Gradient Descent The *Stochastic Gradient Descent* block represents a type of optimizer used in the neural network. It has an Optimizer output port that is linked to the *Compiler* block.

Compiler The *Compiler* block receives in one of its input ports the sequence of layers defined for the neural network using the *Dense* blocks. On the other input port, it receives the optimizer for the neural network. For the *Compiler* block to be valid, the number of layers used must be at least one. Therefore, the *Sequential* block must not be direct linked to the *Compiler* block. It has to have at least one *Dense* block in between. This condition is reflected in the *Compiler* block's assertions shown in Listing 5.25.

```
Input Layer port
   num_dense_blocks >= 1
```

Code Listing 5.25: Compiler block assertions

5.5 Deployment

The front-end and back-end applications are deployed on a web server and compilation server, respectively. The back-end server implements an Application Programming Interface (API) using the Representational State Transfer (REST) architecture to receive type checking and compilation requests over HyperText Transfer Protocol (HTTP).

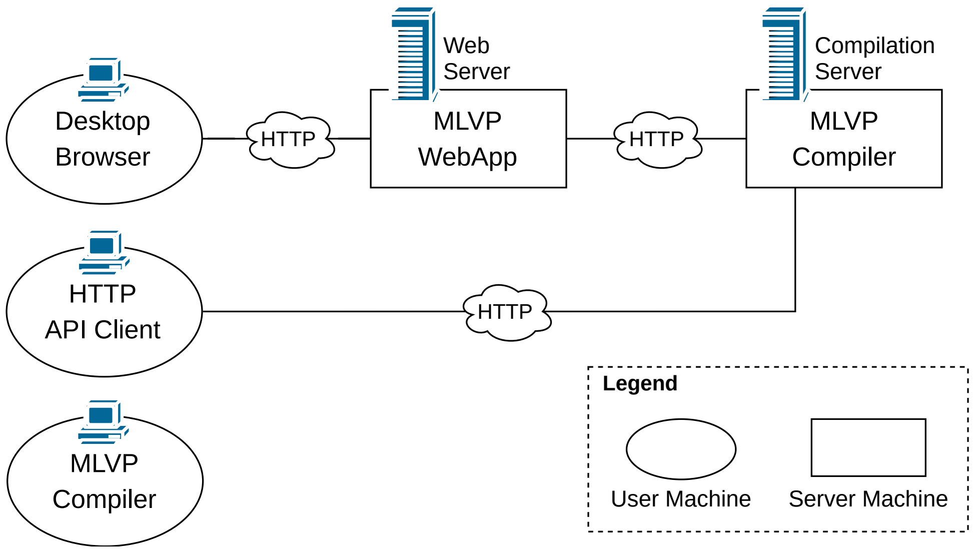


Figure 5.23: MLVP Deployment Allocation View

The requests are handled by different endpoints that make use of the MLVP compiler to send a response. Figure 5.24 demonstrates how each endpoint uses the compiler phases to prepare a response. Recalling what those phases were. The parser phase is where the JSON representation of the pipeline canvas is parsed to recreate the DAG representations of the pipelines within. In the topological sorter phase, the DAG representing the pipeline are sorted topologically. The data flow elaboration phase is where the metadata is successively processed and passed from each port to the next. The type checker phase is where all formulas for each block and link are checked for satisfiability. Finally, in the code generation phase, a valid pipeline is translated into executable source code.

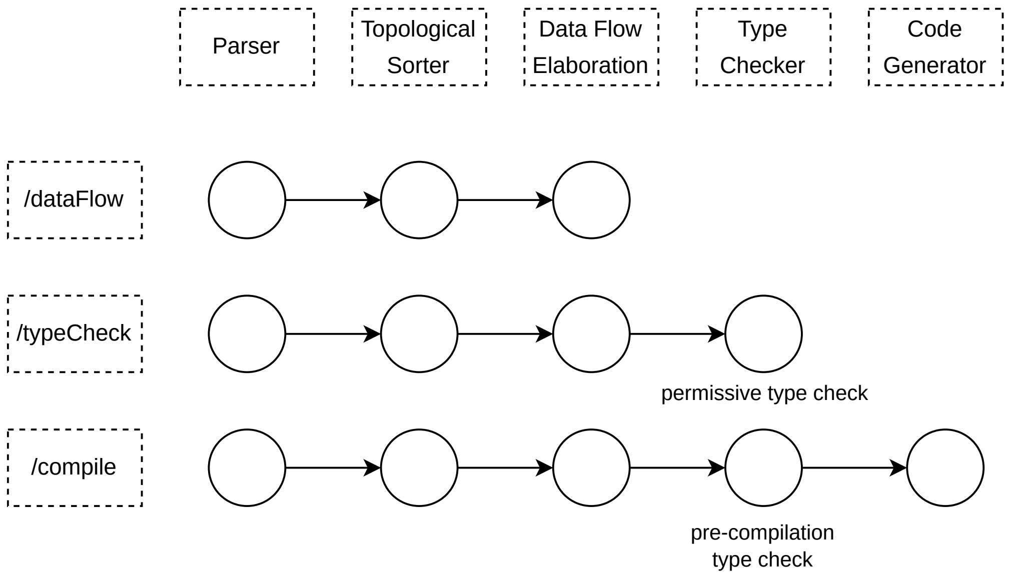


Figure 5.24: Back-end endpoints

The MLVP compiler can be used without the front-end application. Either by directly contacting the compilation server using an HTTP client, such as Postman, to send a POST request with the JSON in its body or by locally passing the JSON path to the compiler application.

Chapter 6

Evaluation

This work aims to detect problems with ML pipelines before execution. The main benefit of this approach is the time and resources saved when a problem is detected early instead of deep into the pipeline.

The tool was evaluated through its ability to detect common errors in ML pipelines. These errors were identified in meetings with Feedzai's data science project managers to ensure that the evaluation focuses on fundamental and realistic errors.

Tables 6.1 and 6.2 contain the dataset metadata of two datasets constructed from real-world data on fraudulent credit card transactions. They are used throughout this section to demonstrate the error detection of the tool along with the credit card fraud dataset from Kaggle presented in Section 2.2. This dataset is available at `https://www.kaggle.com/mlg-ulb/creditcardfraud`.

Column Name	**Column Type**
time	int
bank_id	int
client_id	int
cost	float
class	int

Table 6.1: Dataset A metadata

Column Name	**Column Type**
card_number	int
card_brand	string
amount	float
class	int

Table 6.2: Dataset B metadata

6.1 Dataset concatenation

Dataset concatenation is merging several datasets into a single dataset, either by performing horizontal or vertical concatenation. For the resulting dataset to be valid, intervening datasets' characteristics must be compatible, which is not always the case. Often, data scientists concatenate datasets, whose result is an invalid dataset with duplicate column names or values of both string and numeric values in the same column, leading to errors in the pipeline.

6.1.1 Horizontal Concatenation

Consider that datasets from Tables 6.1 and 6.2 were to be concatenated horizontally. Since Datasets A and B have five and four columns, the final dataset would have nine columns. For the operation to be valid, both datasets would need to: have an equal number of rows, to avoid having columns with empty cells; unique column names so that the final dataset would be free of duplicate column names. However, these datasets share the 'class' column name. Therefore the concatenation would result in an invalid dataset.

Figure 6.1 demonstrates the process of attempting to horizontally concatenate datasets A and B in MLVP. Once loaded into two instances of the *Sample CSV* block, they are linked to a *Horizontal Concatenation* block. Since both datasets share the 'class' column name, when attempting to link the *Sample CSV B* block to the *Horizontal Concatenation* block after having the *Sample CSV A* already linked, an assertion violation is detected, and the link is never created. The assertion violated is shown in Listing 6.1.

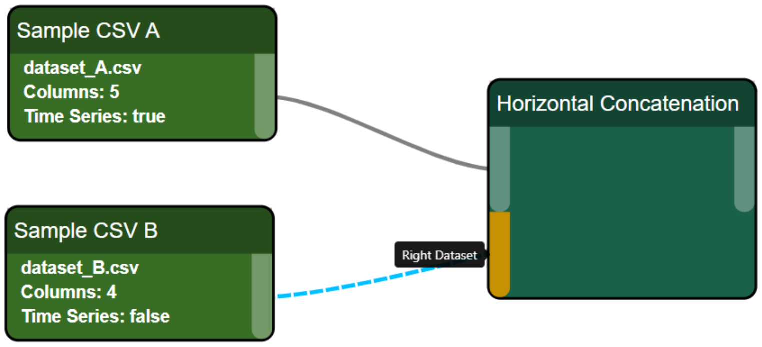


Figure 6.1: Datasets A and B share a column name

```
Horizontal Concatenation
   Property unique_columns_from_input_ports
```

Code Listing 6.1: Horizontal concatenation error message

6.1.2 Vertical Concatenation

Consider that datasets from Tables 6.1 and 6.2 were to be concatenated vertically. For the final dataset to be correct, both A and B datasets would need to have an equal number of columns, which is not the case. Dataset A has five columns and B four. For the operation to be valid, both datasets would need to: have an equal number of columns, to avoid having columns with empty cells; equal column type at each column index from both datasets so that the final dataset would be free of mixed types in all columns, for instance, values of type string and float in each column.

Figure 6.2 demonstrates the process of attempting to vertically concatenate datasets A and B in MLVP. Once the datasets are loaded into two instances of the *Sample CSV* block, they are linked to a *Vertical Concatenation* block. Since datasets A and B do not have an equal number of columns, when attempting to link the *Sample CSV B* block to the *Vertical Concatenation* block after having the *Sample CSV A* already linked, an assertion violation is detected, and the link is never created. The assertion violated is shown in Listing 6.2.

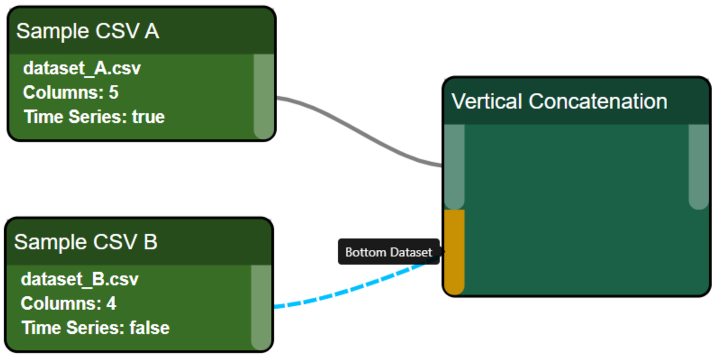


Figure 6.2: Datasets A and B do not have an equal number of columns

```
Vertical Concatenation
   Top Dataset Port n_cols == Bottom Dataset Port n_cols
   Concatenated Dataset Port n_cols == Bottom Dataset Port
      n_cols

Sample CSV A
   Dataset Port n_cols == 5
```

Code Listing 6.2: Vertical concatenation error message

6.2 Dataset transformation

Dataset transformation is when a dataset in its raw state is transformed into modeled data, ready for analysis or training. This process could be, for instance, the conversion of non-numeric features into numeric or the creation of new features from existing ones.

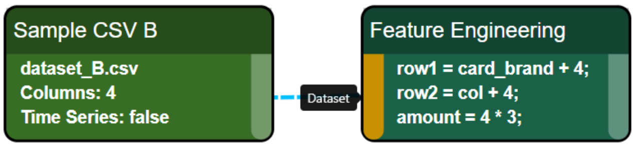
6.2.1 Feature Engineering

Feature engineering is creating new dataset columns from existing ones for better representation of the underlying problem, possibly leading to improved model accuracy.

Consider that dataset B from Table 6.2 is used to engineer new features. For the resulting dataset to be valid, the operands must be of compatible types. For instance, in addition operations, both operands must be of type int or float. When working with unseen large datasets, it is common for data scientists to create new features using incompatible column types unintentionally.

Figure 6.3 demonstrates the process of engineering new features on MLVP using dataset B. Within the *Feature Enginnering* block, three new columns are being created: column 'row1' through an addition operation of the values from column 'card_brand' (of type string) and the literal 4 (of type int); column 'row2' through an addition operation of the values from column 'col' (nonexistent on the dataset) and the literal 4 (of type int); column 'amount' (whose name already existents in the dataset) through a multiplication operation of the literals 4 (of type int) and 3 (of type int);

All three operations contain type errors that MLVP detects. The respective error message is shown in Listing 6.2.



dataset_B.csv
Columns: 4
Time Series: false

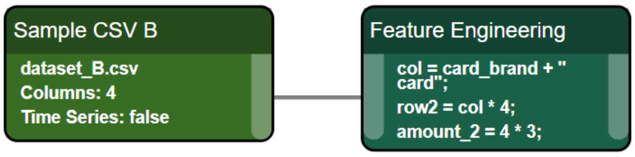
row1 = card_brand + 4;
row2 = col + 4;
amount = 4 * 3;

Figure 6.3: Operations with invalid types and undeclared variables

```
Feature Engineering
   expression using invalid types "string" + "int" - line 1
   column "col" does not exist on the dataset - line 2
   column "amount" is already part of the dataset - line 3
```

Code Listing 6.3: Feature engineering error message

The errors shown in Listing 6.2 could be fixed respectively by using compatible types on the addition operation, creating the column col before attempting to use it, and choosing a name for the new column that does not exist on the dataset, respectively.



dataset_B.csv
Columns: 4
Time Series: false

col = card_brand + "
card";
row2 = col * 4;
amount_2 = 4 * 3;

Figure 6.4: All operations use valid types and declared variables

6.2.2 Temporal Aggregation

A dataset is considered a time series when all data points are ordered chronologically. In the case of a credit card transaction dataset, each row would represent a transaction in time. Temporal aggregation operations require datasets to represent a time series, for the result of applying a rolling window on a feature of the dataset reflects meaning on how the data changes over time. For instance, it would detect if a given transaction was ten times larger than the mean of the past month. The result of applying temporal aggregation operations on datasets that do not represent a time series is meaningless. Therefore they should not be done.

Consider that dataset B from Table 6.2 does not represent a time series. Figure 6.5 demonstrates the process of temporal aggregating the dataset B, which is loaded into *Sample CSV B* block, by linking it to the *Temporal Agregation* block. Since the *Sample CSV B* block does not represent a time series, the port property reflects that as False. The *Temporal Agregation* block by expecting the property to be True causes a contradiction in the assertions. Both assertions are shown in Listing 6.4.

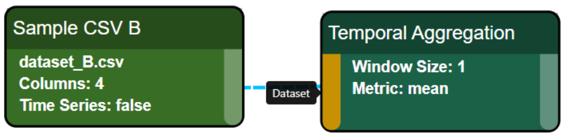


Figure 6.5: Dataset B does not represent a time series

```
Sample CSV B
   Dataset Port time_series == False

Temporal Aggregation
   Dataset Port time_series == True
```

Code Listing 6.4: Temporal aggregation error message

The solution for this problem would be to identify the dataset as being a time series. This identification could be manual or eventually automatic when a sample of the dataset is present.

6.3 Imbalanced Datasets

The prediction accuracy of models trained with imbalanced datasets may be affected due to the training data lacking some target variable. Furthermore, in real scenarios, the variable that was scarce in training may appear in more significant quantities, thus affecting the model performance.

Consider the credit card dataset from Kaggle, which is heavily imbalanced. The dataset was loaded into the *Import from CSV* block, which was used to train a classifier by linking it to the *Random Forest Classifier* block. Since the dataset is imbalanced, MLVP detects it and does not allow the formation of the link.Listing 6.5 contains the errors shown when attempting to linked both blocks.

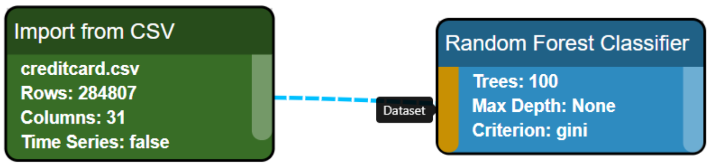


Figure 6.6: Using an imbalanced dataset to train the classifier

```
Import from CSV
   Dataset Port balanced == False

Random Forest Classifier
   Dataset Port balanced == True
```

Code Listing 6.5: Imbalanced dataset error message

The solution for this issue would be to either have an *Undersampling*, or *Oversampling* block linked between the *Import From CSV* and *Random Forest Classifier* blocks (Figure 6.7). The under-sampling and over-sampling techniques consist of deleting dataset rows in the majority class and duplicating rows in the minority class to even out all class instances in the dataset. Therefore, the requirement of receiving a balanced dataset in the *Random Forest Classifier* block is ensured.

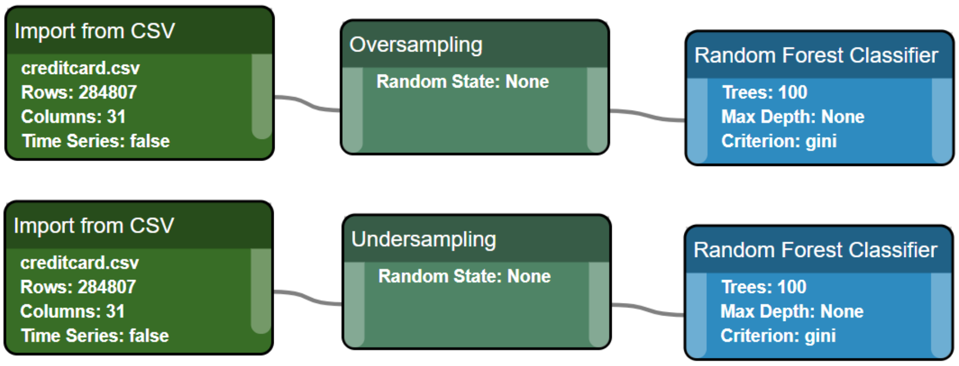


Figure 6.7: Balancing the dataset before using it for training

6.4 Processing Datasets

When a dataset is fed into a ML pipeline, it can go through many operations. For example, new columns are engineered, others are encoded, and rows are discarded from the dataset. The order in which operations occur may impact the pipeline execution time.

Consider the credit card dataset from Kaggle [52], which contains 284807 rows, goes first through feature engineering and then an undersampling operation. The resulting dataset would be the same as doing the undersampling operation first and then the feature engineering. However, the work applied and the time consumed to achieve the same result are higher on the second approach. Since the under-sampling operation balances a dataset by removing the majority classes to even out all instanced in a dataset, having the dataset go through the feature engineering first leads to the discard of calculation on the under-sampling operation. Therefore, for the best efficiency, the under-sampling should come before the feature engineering operation. For instance, every operation that processes the whole dataset, feature engineering or encoding operations, should always come before a reducing operation (undersampling or sampling).

Figure 6.8 demonstrates the scenario described in MLVP, in which the credit card dataset is loaded into the *Import from CSV* block, then linked to a *Feature Engineering* block. When attempting to create a link with the *Undersampling* block, the tool detects that the order of operations results in unnecessary consumption of resources, and the link is not created. Listing 6.6 shows the error message displayed when attempting to perform such an operation.

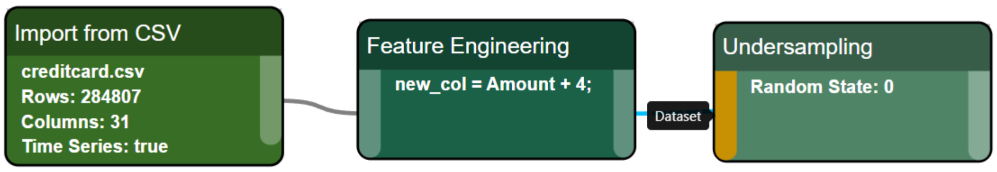


Figure 6.8: Order of operations results in unnecessary consumption of resources.

```
   Undersampling
      Dataset Port processed == False

   Feature Engineering
      Engineered Dataset Port processed == True
```

Code Listing 6.6: Processing dataset error message

The correct approach that does not trigger any error is to have the *Undersampling* block linked before the *Feature Engineering* block. Figure 6.9 demonstrates the solution.

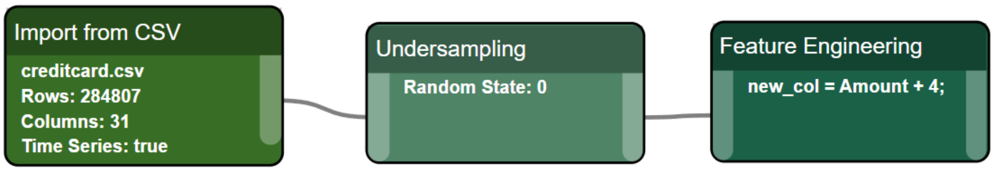


Figure 6.9: Optimized order of operations

Chapter 7

Conclusion

Code-based approaches to build ML pipelines rely on the programming proficiency of the data scientist. VP advancements have been made to facilitate the task of creating ML pipelines. Still, they come with limitations. The coupling of the tool's pipeline development and execution environments compromises the high-performance execution of the pipeline on clusters, and the tool lock-in prevents the migration of the pipeline to other alternatives.

Our proposed tool (MLVP) improves over existing VPL for creating ML pipelines by approaching the creation process from a static point of view. Each block contains a set of assertions using dataset metadata to verify pipeline validity. During type checking, the addition of all assertions into an SMT solver dictates pipeline satisfiability. Once a pipeline is proven to be valid, the MLVP tool compiles the pipeline into executable code, allowing for the execution of the pipeline in any machine (e.g., clusters).

This work proved that ML pipeline verifications could be approached using design by contract. The main principles of this model as documented by Reddy [53] are threefold:

- Pre-conditions: the client is obligated to meet a function's required preconditions before calling a function. If the preconditions are not met, then the function may not operate correctly;

- Post-conditions: the function guarantees that certain conditions will be met after it has finished its work. If a post-condition is not met, then the function did not complete its work correctly;

- Class invariant: constraints that every instance of the class must satisfy. This defines the state that must hold true for the class to operate according to its design.

In MLVP, the pre-conditions are the assertions regarding the dataset received from the blocks' input port. The post-conditions are the characteristics ensured about the dataset leaving the block's output port. Finally, the class invariant constraints are the assertions regarding the block's properties. In this work we applied design by contract in a VP approach to orchestrate ML pipelines. Applying it in code-based approaches could also be explored in the future.

Although MLVP achieved advances in the creation and execution of ML pipelines, there are still improvement opportunities, for example:

- Enrich the VPL by implementing more blocks;
- Autocomplete new links, based on compatible blocks;
- Automatic optimization of the pipeline's DAG representation;
- Predict pipeline execution time;

The development of MLVP resulted in the design and implementation of a VPL to build ML pipelines featuring: 29 blocks for data science and ML operations; a type checker to statically verify pipeline semantic and syntactic correctness; a compiler to compile the VPL into executable code.

Bibliography

[1] G. Tzanis, I. Katakis, I. Partalas, and I. Vlahavas, “Modern applications of machine learning,” 01 2006. [Online]. Available: https://www.researchgate.net/publication/228340464_Modern_Applications_of_Machine_Learning

[2] D. S. W. Ho, W. Schierding, M. Wake, R. Saffery, and J. O’Sullivan, “Machine learning SNP based prediction for precision medicine,” *Frontiers in genetics*, vol. 10, pp. 267–267, Mar 2019, 30972108[pmid]. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/30972108

[3] J. Jurgovsky, M. Granitzer, K. Ziegler, S. Calabretto, P.-E. Portier, L. He-Guelton, and O. Caelen, “Sequence classification for credit-card fraud detection,” *Expert Systems with Applications*, vol. 100, pp. 234–245, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417418300435

[4] Z. Wang and M. O’Boyle, “Machine learning in compiler optimization,” *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, 2018.

[5] K. Feldman, L. Faust, X. Wu, C. Huang, and N. V. Chawla, “Beyond volume: The impact of complex healthcare data on the machine learning pipeline,” in *Towards Integrative Machine Learning and Knowledge Extraction*, A. Holzinger, R. Goebel, M. Ferri, and V. Palade, Eds. Cham: Springer International Publishing, 2017, pp. 150–169. [Online]. Available: https://arxiv.org/abs/1706.01513

[6] M. W. Libbrecht and W. S. Noble, “Machine learning applications in genetics and genomics,” *Nature Reviews Genetics*, vol. 16, no. 6, pp. 321–332, Jun 2015. [Online]. Available: https://doi.org/10.1038/nrg3920

[7] T. H. Davenport and D. Patil, “Data scientist: The sexiest job of the 21st century,” 10 2012. [Online]. Available: https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century

[8] P. Pereira, J. Cunha, and J. P. Fernandes, “On understanding data scientists,” in *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2020, pp. 1–5.

[9] M. Kim, T. Zimmermann, R. DeLine, and A. Begel, “Data scientists in software teams: State of the art and challenges,” *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1024–1038, 2018.

[10] M. B. Kery, A. Horvath, and B. Myers, *Variolite: Supporting Exploratory Programming by Data Scientists*. New York, NY, USA: Association for Computing Machinery, 2017, p. 1265–1276. [Online]. Available: https://doi.org/10.1145/3025453.3025626

[11] E. Breck, M. Zinkevich, N. Polyzotis, S. Whang, and S. Roy, "Data validation for machine learning," in *Proceedings of SysML*, 2019. [Online]. Available: https://mlsys.org/Conferences/2019/doc/2019/167.pdf

[12] P. Pandey, "Data preprocessing: Concepts," Nov 2019. [Online]. Available: https://towardsdatascience.com/data-preprocessing-concepts-fa946d11c825

[13] Google, "Common ml problems," May 2021. [Online]. Available: https://developers.google.com/machine-learning/problem-framing/cases

[14] M. Eric, "A machine learning primer." [Online]. Available: https://www.confetti.ai/assets/ml-primer/ml_primer.pdf

[15] S. Asiri, "Machine learning classifiers," Jun 2018. [Online]. Available: https://towardsdatascience.com/machine-learning-classifiers-a5cc4e1b0623

[16] A. Al-Masri, "What are overfitting and underfitting in machine learning?" Jun 2019. [Online]. Available: https://towardsdatascience.com/what-are-overfitting-and-underfitting-in-machine-learning-a96b30864690

[17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. [Online]. Available: https://scikit-learn.org/

[18] E. Frank, M. A. Hall, G. Holmes, R. B. Kirkby, B. Pfahringer, and I. H. Witten, *Weka: A machine learning workbench for data mining*, ser. Data Mining and Knowledge Discovery Handbook. Springer, 2005, pp. 1305–1314, 62. [Online]. Available: https://hdl.handle.net/10289/1497

[19] H2O.ai, *H2O: Scalable Machine Learning Platform*, 2020, version 3.30.0.6. [Online]. Available: https://github.com/h2oai/h2o-3

[20] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on

heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[21] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939785

[22] N. Baccouri, "Machine learning without writing code," Oct 2020. [Online]. Available: https://medium.com/swlh/machine-learning-without-writing-code-984b238dd890

[23] a. M. c. QuantumBlack, "Introducing kedro: The open source library for production-ready machine learning code," Jun 2019. [Online]. Available: https://medium.com/quantumblack/introducing-kedro-the-open-source-library-for-production-ready-machine-learning-code-d1c6d26ce2cf

[24] N. Schrock, "Moving past airflow: Why dagster is the next-generation data orchestrator," May 2021. [Online]. Available: https://medium.com/dagster-io/moving-past-airflow-why-dagster-is-the-next-generation-data-orchestrator-e5d297447189

[25] J. Demšar, T. Curk, A. Erjavec, Črt Gorup, T. Hočevar, M. Milutinovič, M. Možina, M. Polajnar, M. Toplak, A. Starič, M. Štajdohar, L. Umek, L. Žagar, J. Žbontar, M. Žitnik, and B. Zupan, "Orange: Data mining toolbox in python," *Journal of Machine Learning Research*, vol. 14, pp. 2349–2353, 2013. [Online]. Available: https://orangedatamining.com/

[26] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, C. Sieb, K. Thiel, and B. Wiswedel, "Knime: The konstanz information miner," in *Data Analysis, Machine Learning and Applications*, C. Preisach, H. Burkhardt, L. Schmidt-Thieme, and R. Decker, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 319–326.

[27] O. Petashchuk, "Datrics: the fastest way to create and operate ml," Apr 2020. [Online]. Available: https://medium.com/datrics-ai-launches-a-free-alpha-version-of-our/datrics-ai-launches-a-free-alpha-version-of-data-science-platform-for-smes-2c2995e80365

[28] J. Souyris, V. Wiels, D. Delmas, and H. Delseny, "Formal verification of avionics software products," in *FM 2009: Formal Methods*, A. Cavalcanti and D. R. Dams, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 532–546.

[29] C. Urban and A. Miné, "A review of formal methods applied to machine learning," 2021.

[30] Z. Kurd and T. Kelly, "Establishing safety criteria for artificial neural networks," in *Knowledge-Based Intelligent Information and Engineering Systems*, V. Palade, R. J. Howlett, and L. Jain, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 163–169.

[31] F. Ranzato and M. Zanella, "Robustness verification of support vector machines," 2019.

[32] A. Kantchelian, J. D. Tygar, and A. D. Joseph, “Evasion and hardening of tree ensemble classifiers,” 2016.

[33] H. Chen, H. Zhang, S. Si, Y. Li, D. Boning, and C.-J. Hsieh, “Robustness verification of tree-based models,” 2019.

[34] N. SATO, H. KURUMA, Y. NAKAGAWA, and H. OGAWA, “Formal verification of a decision-tree ensemble model and detection of its violation ranges,” *IEICE Transactions on Information and Systems*, vol. E103.D, no. 2, p. 363–378, Feb 2020. [Online]. Available: http://dx.doi.org/10.1587/transinf.2019EDP7120

[35] G. Einziger, M. Goldstein, Y. Sa’ar, and I. Segall, “Verifying robustness of gradient boosted models,” 2019.

[36] J. Törnblom and S. Nadjm-Tehrani, “Formal verification of random forests in safety-critical applications,” in *Formal Techniques for Safety-Critical Systems*, C. Artho and P. C. Ölveczky, Eds. Cham: Springer International Publishing, 2019, pp. 55–71.

[37] ——, “An abstraction-refinement approach to formal verification of tree ensembles,” in *Computer Safety, Reliability, and Security*, A. Romanovsky, E. Troubitsyna, I. Gashi, E. Schoitsch, and F. Bitsch, Eds. Cham: Springer International Publishing, 2019, pp. 301–313.

[38] ——, “Formal verification of input-output mappings of tree ensembles,” *Science of Computer Programming*, vol. 194, p. 102450, Aug 2020. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2020.102450

[39] F. Ranzato and M. Zanella, “Abstract interpretation of decision tree ensemble classifiers,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, p. 5478–5486, 2020.

[40] S. Calzavara, P. Ferrara, and C. Lucchese, “Certifying decision trees against evasion attacks by program analysis,” 2020.

[41] C. Urban and P. Müller, “An abstract interpretation framework for input data usage,” in *Programming Languages and Systems*, A. Ahmed, Ed. Cham: Springer International Publishing, 2018, pp. 683–710.

[42] C. Urban, “What programs want: Automatic inference of input data specifications,” 2020.

[43] B. C. Pierce, *Types and programming languages*. The MIT Press, 2002.

[44] TypeScript is an open-source language which builds on javascript. [Online]. Available: https://www.typescriptlang.org/

[45] React, a javascript library for building user interfaces. [Online]. Available: https://reactjs.org/

[46] Rete is a modular framework for visual programming. [Online]. Available: https://rete.js.org/

[47] Beautiful React Diagrams, a tiny collection of lightweight react components for building diagrams with ease. [Online]. Available: https://github.com/beautifulinteractions/beautiful-react-diagrams

[48] Storm React Diagrams, a flow and process orientated diagramming library inspired by blender, labview and unreal engine. [Online]. Available: https://github.com/projectstorm/react-diagrams

[49] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-78800-3_24

[50] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. Raleigh, NC: Pragmatic Bookshelf, 2013. [Online]. Available: https://www.safaribooksonline.com/library/view/the-definitive-antlr/9781941222621/

[51] ANTLR is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. [Online]. Available: https://www.antlr.org/

[52] Kaggle Credit Card Fraud Detection Dataset. [Online]. Available: https://www.kaggle.com/mlg-ulb/creditcardfraud

[53] M. Reddy, "Chapter 9 - documentation," in *API Design for C++*, M. Reddy, Ed. Boston: Morgan Kaufmann, 2011, pp. 267–289. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123850034000099

[54] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. The MIT Press, 2016.

[55] S. Murthy, "Automatic construction of decision trees from data: A multi-disciplinary survey," *Data Min. Knowl. Disc.*, vol. 2, 03 2000.

[56] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001. [Online]. Available: https://doi.org/10.1023/a:1010933404324

[57] J. H. Friedman, "Greedy function approximation: A gradient boosting machine." *The Annals of Statistics*, vol. 29, no. 5, pp. 1189 – 1232, 2001. [Online]. Available: https://doi.org/10.1214/aos/1013203451

[58] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000.

[59] Z3 is a theorem prover from microsoft research. [Online]. Available: https://github.com/Z3Prover/z3

[60] Weka is a collection of machine learning algorithms for data mining tasks. [Online]. Available: https://www.cs.waikato.ac.nz/ml/weka/