Handling of Schema Evolution in Machine Learning Pipelines

A Thesis

submitted to the designated
by the Assembly
of the Department of Computer Science and Engineering
Examination Committee

by

Athanasios Paligiannis

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN DATA AND COMPUTER SYSTEMS ENGINEERING

WITH SPECIALIZATION
IN ADVANCED COMPUTER SYSTEMS

University of Ioannina
School of Engineering
Ioannina 2022

Examining Committee:

- **Panos Vassiliadis**, Professor, Department of Computer Science and Engineering, University of Ioannina (advisor)
- **Evaggelia Pitoura**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Apostolos Zarras**, Professor, Department of Computer Science and Engineering, University of Ioannina

Dedication

To my lovely family.

Table of Contents

List of Figures ii

List of Tables iii

List of Algorithms iv

Abstract v

Εκτεταμένη Περίληψη vii

1 Introduction 1

1.1 Schema Evolution 1

1.2 Machine Learning 3

1.2.1 Types of Data in Machine Learning 3

1.2.2 Types of Machine Learning 4

1.3 Approach to handling schema evolution in ML pipelines 5

1.3.1 Steps of Analysis 5

2 Related Work 7

3 Problem Definition 12

3.1 What is a machine learning pipeline ? 12

3.2 What kind of stages can be in Machine learning pipeline? 13

3.3 Formal description and problem formulation 15

3.4 Framework used in development of machine learning pipeline 16

3.5 The MLlib library 17

3.6 Spark's API for Data management 18

3.7 How changes in external storage affect machine learning pipeline? . . . 19

4 Pipeline Abstraction and Evolution 20
4.1 Abstract Syntax Trees 20
4.2 Eclipse JDT 22
4.3 What is the Java Model of Eclipse JDT? 22
4.4 Eclipse Plugin Projects 23
4.5 How to find dependencies in ML Pipeline? 24
4.6 Correctness requirements and Spark particularities 28
4.7 Modeling of the information extracted from ASTParser via graph diagrams 30
4.7.1 The Name-Dependency Graph 31
4.7.2 Topological sorting 34
4.8 How schema changes can be detected 35
5 Experiments 39
5.1 Time Metrics during AST Parser Execution 39
5.1.1 AST Parser Performance over the Number of Source Code Lines 41
5.1.2 AST Parser Performance over the Number of Source Code Files 42
5.1.3 AST Parser Performance over the Number of Pipeline Stages . . 43
5.1.4 AST Parser Performance over the Number of Pipelines 44
5.2 Time Metrics for the GraphAnalyzer Execution 44
5.2.1 Time performance of GraphAnalyzer over the number of Pipelines 45
5.2.2 Time performance of GraphAnalyzer over the number of Pipeline Stages 46
5.2.3 Time performance of GraphAnalyzer over the Type of Graph . . 47
6 Conclusions and Future Work 49
Bibliography 51

List of Figures

1.1 History of schema. 2

2.1 Mapping Studies Process [4] 8
2.2 The homepage of dblp 9

3.1 An example of a Machine Learning Pipeline[5] 12
3.2 An example of a StopWordsRemover stage 14
3.3 An example of a Estimator stage 14
3.4 Apache Spark Architecture[6] 16
3.5 MLlib library [6] 17

4.1 Stages of Compiler. 21
4.2 Abstract Syntax Tree Example 21
4.3 Eclipse JDT. 22
4.4 JavaModel Structure 23
4.5 Eclipse Runtime Application. 24
4.6 Pipeline Statement. 25
4.7 Flow diagram recognize Pipeline Stage. 26
4.8 Pipeline Stage Statement. 27
4.9 Pipeline Stage Statement with non-typical keywords. 27
4.10 An example of file exported by ASTParser Application 28
4.11 Architecture of Application 28
4.12 Entities of GraphAnalyzer 30
4.13 Flow diagram which show how to recognize edge between nodes. 32
4.14 Graph with no external dependencies. 33
4.15 Flow diagram which show how to recognize edges comes from external data source 33

4.16 Graph with external dependencies. 34
4.17 System checks for missing dependencies 36
4.18 Graph before schema change 37
4.19 Graph after schema change 38

5.1 Time metrics based on Number of Source Code Lines 41
5.2 Time metrics based on Number of Source Code Files 42
5.3 Time metrics based on Number of Pipeline Stages 43
5.4 Time metrics based on Number of Pipelines 44
5.5 Time metrics based on Number of Pipelines 45
5.6 Time metrics based on Number of Pipeline Stages. 46
5.7 Time metrics based on Type of Graph 47
5.8 Types of Graph [13] 47

List of Tables

2.1 Compilation of the corpus of papers. 10

3.1 An example of dependency to schema attributes 19

List of Algorithms

4.1 Topological Sorting of Graph 35

Abstract

Athanasios Paligiannis, M.Sc. in Data and Computer Systems Engineering, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, 2022.
Handling of Schema Evolution in Machine Learning Pipelines.
Advisor: Panos Vassiliadis, Professor.

Recently, data analysis and machine learning are increasing rapidly, as they are widely used in industry. *Machine learning pipelines* is a new trend in artificial intelligence whose purpose is to train automatically the model based on a certain input, via a succession of machine learning and data transformation steps.In this context, Apache Spark with its MLib library, is a powerful tool for data analysis, used to train models via machine learning pipelines. The main idea of machine learning pipelines is to compose a sequence of steps, called *stages* into a linear workflow that ultimately either trains or fits a model to the incoming data, and can be later deployed to work with more incoming data. These pipelines are constructed programmatically, in a host language (in our case: Java) and allow the automated execution of the entire pipeline via a single program invocation.

The current thesis deals with machine learning pipelines from the viewpoint of software engineering, and provides two contributions. First, we take the source code of the pipelines and provide an abstraction of it, as main-memory structures. This is achieved by exploiting the Abstract Syntax Tree of the source code and extracting the necessary statements that define stages and pipelines, out of the entire set of statements that constitute the source code. This is facilitated by exploiting the Eclipse JDT API, which is tailored exactly for analyzing source code via its Abstract Syntax Tree. The outcome of this process is a main-memory representation of the pipeline as a graph, with stages as its nodes and input-output dependencies between subsequent

stages as its edges. This also allows the automation of the graphical representation of all machine learning pipelines found in the source code of a certain project.

Second, by exploiting this graph representation, we can assess the impact of schema evolution in a set of pipelines that all stem from the same data provider (e.g., a data file with structured records). Assuming that the structure of the records of a file is changed, with certain attributes being inserted and other attributes being deleted with respect to its previous version, we exploit the dependencies that exist between stages of pipeline and the source data, and propagate the impact of the change over the graph. Moreover, we also visualize the impact of the change in the graphical representation of the pipeline.

Εκτεταμενη Περιληψη

Αθανάσιος Παληγιάννης, Δ.Μ.Σ. στη Μηχανική Δεδομένων και Υπολογιστικών Συστημάτων, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, 2022.
Διαχείριση εξέλιξης σχήματος σε ακολουθίες μηχανικής μάθησης.
Επιβλέπων: Παναγιώτης Βασιλειάδης, Καθηγητής.

Στο προγραμματιστικό περιβάλλον Apache Spark, μια διοχέτευση μηχανικής μάθησης (machine learning pipeline) είναι μια ροή εργασίας που χρησιμοποιείται για να παράγει ένα μοντέλο μηχανικής μάθησης μέσω σειράς επί μέρους βημάτων τα οποία συντίθεται για την παραγωγή του τελικού αποτελέσματος. Η σύνθεση αυτή γίνεται προγραμματιστικά, μέσω των σχετικών τύπων δεδομένων που προσφέρει το περιβάλλον Spark. Λόγω της δημοφιλίας του περιβάλλοντος Spark, οι διοχετεύσεις αυτές θα γίνονται όλο και πιο ευρέως χρησιμοποιούμενες στο μέλλον.

Ο στόχος της παρούσας εργασίας δεν αφορά την ουσία της μηχανικής μάθησης, αλλά την αντιμετώπιση της ενσωμάτωσης της ροής εργασίας προγραμματικά εντός του πηγαίου κώδικα (καθώς ένα σύνολο από διοχετεύσεις μηχανικής μάθησης ορίζεται και συντίθεται με προγραμματιστικό τρόπο ο οποίος δεν ακολουθεί υποχρεωτικά κάποια συγκεκριμένη δομή). Συγκεκριμένα, ο στόχος της εργασίας είναι η επεξεργασία πηγαίων αρχείων κώδικα, με στόχο μια αφαιρετική τους αναπαράσταση. Ο μηχανισμός που εισάγουμε είναι ένας μηχανισμός επεξεργασίας των αρχείων που οδηγεί σε μια αναπαράσταση Abstract Syntax Trees, γνωστά από το χώρο των μεταφραστών λογισμικού. Η αναπαράσταση του κώδικα σε ένα υψηλότερο επίπεδο αφαίρεσης, επιτρέπει τη διαχείρισή του ως ένα γράφημα, με κόμβους τα επί μέρους βήματα της ροής εργασίας και ακμές τις διοχετεύσεις δεδομένων.

Η γραφοθεωρητική αυτή αναπαράσταση, στη συνέχεια επιτρέπει την εκμετάλλευσή της ποικιλοτρόπως. Στην εργασία αυτή εξετάζουμε το πρόβλημα της εξέλιξης του σχήματος, όπου το πηγαίο αρχείο αντικαθίσταται από μια νέα εκδοχή του, με

αλλαγμένη εσωτερική δομή, και η αφαιρετική αναπαράσταση της ροής εργασίας επιτρέπει τον εντοπισμό των σημείων εκείνων της ροής που γίνονται συντακτικώς εσφαλμένα λόγω της αλλαγής αυτής.

Chapter 1

Introduction

1.1 Schema Evolution

1.2 Machine Learning

1.3 Approach to handling schema evolution in ML pipelines

1.1 Schema Evolution

Before thousands of years Heraclitus said that everything is constantly changing. Today we have to admit that this is a fact that applies in every sector of our life. Especially in computer science , changes are tremendous the last decades. Changes are taking place in hardware and to software too. There is a strong connection between these fields and many times one of them enforce the other to evolve to support the changes.

Our focus estimates only on changes in schema of relational database. As it is common, schema of relational database is rarely static. Many reasons lead database administrators to adjust their schema and create a new version of it. One of them, may be the update of Database Management System (DBMS) and changing schema is the only option so as the database to be still functional.

Other frequent reason of schema evolution is related with changes in software that also affects the database. For example is very common to industry, in new releases of products, new features to be added or bugs to be fixed. Many of these features

require new changes in the database. Some of them may require only new imports of data in database.

But there are features that require changes in tables of database (add new column, delete current column, rename current column) or changes in data model of database. So in these cases database administrator should adjust the current schema and create a new version of schema. [1]

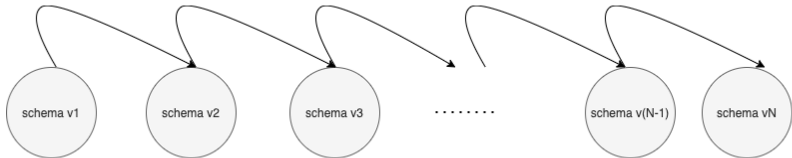

Figure 1.1: History of schema.

As it is obvious from figure 1.1 schema version follows a sequential order during evolution of it. From transaction v(n-1) to v(n) is very important to care about impact effects of these changes firstly to data and secondly to source code.

Direct impacts of schema changes are inside of Data of Database. After every schema change it is very important to ensure about validity of data and correctness of data model. There are many scenarios that can change the schema. For example a new table can be added to the current schema, or a new column to be added to the current table of schema. These changes mentioned can be handled in an easy way by database administrator. These changes have a small impact in the whole application that can be handled in a convenient way with not too much effort.

There are scenarios that can change the schema too but may change the whole structure of database. Let's assume that we want to just delete a table which is connected with many other tables in database. This change is very risky about our data in database but also may have silent impacts in the source code. In industry we face sometimes situations like this where changes in database can cause silent impacts in source code of applications and program will fail. Purpose of this study is focused in this way to handle schema changes and how we can automatically detect these changes to source code. We have choose a specific field of computer science that is growing rapidly the last years to study about schema changes. This field is machine learning, where the data are using from machine learning procedures changes too and have many similarities with data "live" inside relation database management system.

There is 1 basic term need to be mentioned because they are strongly connected with the topic that we are going to analyze. This term is called machine learning, may is not related directly with schema evolution, but our topic analyze how changes of data affects the processes inside machine learning.

1.2 Machine Learning

Machine learning as term refers to automated detection of meaningful patterns in data. Nowadays machine learning is used widely in every sector of our life that requires information extraction of large data sets. Search engine like Google use algorithms based on machine learning to bring us the best results. Also anti-spam software learns to filter email messages and exclude the spam emails. Digital cameras of smartphones can detect faces and also smartphones trained to recognize voice commands. Machine learning programs are related with ability to learn and adapt.

Machine learning algorithms have the ability to improve themselves through training.[2]

1.2.1 Types of Data in Machine Learning

There are 2 kinds of data in machine learning

- Labeled Data

  Labeled Data means data that have both the input and output parameters in a completely machine-readable pattern. To add label in data needs extra effort from human. An example of adding data is to indicate if an image contains a cor or bike.

- Unlabeled Data

  In opposite direction of Labeled data there are unlabeled data that only have one or none of the parameters in a machine-readable form. It is not need extra effort for human to add the label but needs more complex solutions when this type of data used by machine learning algorithm. [2]

1.2.2 Types of Machine Learning

Depends on the data, uses the machine learning algorithm we have 3 different categories of machine learning.

1. Supervised Learning

   In this type, the machine learning algorithm is trained on labeled data. It can be very powerful when used in the right cases.In supervised learning, the ML algorithm is given a small training dataset to work with. This training dataset is a smaller part of the bigger dataset and serves to give the algorithm a basic idea of the problem, solution, and data points to be dealt with. The training dataset is also very similar to the final dataset in its characteristics and provides the algorithm with the labeled parameters required for the problem.[2]

2. Unsupervised Learning

   Unsupervised machine learning holds the advantage of being able to work with unlabeled data.However, unsupervised learning does not have labels to work off of, resulting in the creation of hidden structures. Relationships between data points are perceived by the algorithm in an abstract manner, with no input required from human beings. The creation of these hidden structures is what makes unsupervised learning algorithms versatile. Instead of a defined and set problem statement, unsupervised learning algorithms can adapt to the data by dynamically changing hidden structures.[2]

3. Reinforcement Learning

   Reinforcement learning directly takes inspiration from how human beings learn from data in their lives. It features an algorithm that improves upon itself and learns from new situations using a trial-and-error method. Favorable outputs are encouraged or ‘reinforced’, and non-favorable outputs are discouraged or ‘punished’.[2]

1.3 Approach to handling schema evolution in ML pipelines

Main purpose of our study is to analyze impacts when schema of dataframe changes and cause problems to ML pipelines. ML pipelines are defined inside source code. More specifically every pipeline consists of collections of steps. Each step can takes a number of arguments as inputs and also can exports a number of arguments as outputs. In all cases this number is greater than or equal to zero (>=0). There are some steps that inputs comes from dataframe. These inputs feed the pipelines, but also are the cause that when schema change pipeline failed because there is a strong connection between dataframe and pipelines.

As it is obvious from the above description, our approach is focused to analysis of the source code and detect these segments in code. Our IDE tool used in this analysis is the famous Eclipse. Eclipse provides the Eclipse JDT API that based on Abstract Syntax Tree and can dive deeply into the analysis of source code. Also inside Eclipse JDT API exists Java Model that can make analysis about the project structure where source code exists. With this powerful tool we are able just to import a ML project that contains the ML pipeline and analyze it.

1.3.1 Steps of Analysis

Firstly we are getting all projects that have been loaded to the workspace. All projects independent of the type will be available to us. Secondly we are filter all these projects so as to keep only java nature projects, projects that source code is written in java language.

In our application, user is able to choose one from available projects for analysis. So a menu with available java projects is shown and user choose a project for the source analysis. In next step, looks in the source packages of project and keeps only packages that contains source code. For every source package we are getting only the class files(ICompilationUnits) contained in that.

At this point we have isolated the class files (ICompilationUnits) from the whole project. All required information is inside of their. With help of Abstract Syntax Tree we are getting all method declarations contains in every class and looking for ML pipeline definitions. Inside pipeline definition, there are contained the names of stages participated in. So we are looking in the same class for all VariableDeclarationStatements. If every VariableDeclarationStatement has the same name and it's

type is kind of Pipeline Stage is regarded as stage and we are looking about inputs, outputs arguments in this statement.

With this way, we have extract from source code all information regarding the ml pipelines exists in project, which stages are participated in every ml pipeline and which are the inputs and outputs arguments of every pipeline stage. All information is been extracted to a specific file called "stages.txt" and based on this file we are going to design the final graph that shows all interactions between stages of pipeline and dataframe.

In the design of graph, every node represents a specific stage of pipeline. Because can exists many pipelines to project, to distinguish in which pipeline node belongs to, every node has a specific identity that is related with pipeline belongs to. So the creation of nodes in graph is not a difficult topic as we know all stages and in which pipeline stage is belongs to each of them . The difficult part was the right placement of edges. To simplify the procedure we split it in two parts. In first part we place only the edges in nodes that belongs in the same pipeline, only adding the interactions between stages of pipeline. In second part we place every edge starts from dataframe and ends in every node of every pipeline, these edges are the external dependencies between pipeline and dataframe.

So every time schema change we focus only in the edges start from dataframe. Inside edge there is the information about which are the borders (startNode, endNode). Looking in endNode we compare inputs arguments with columns of dataframe. If there is not any column to match to the input argument of endNode it means that there is a missing dependency and pipeline will fail. Knowing about which node has the problem we also know which pipeline will fail and we color red ,instead of green that is ok, all nodes of pipeline to indicate the problem.

Chapter 2

Related Work

The basis of this Chapter is to summarize the effort to survey the related work in the area of Schema Evolution in the latest years. This effort took place in the context of the Graduate Program of the Dept. of Computer Science and Engineering in the Univ. of Ioannina and is presented in detail in [3]. The coverage of the related work in the area of schema evolution is reported as a mapping study in [3].

Mapping studies is a special kind of surveying that cares about meta data characteristics of publications that can lead us to useful observations via a very principled process of collecting and classifying bibliography. From this point of view, they differ from traditional surveys, as mapping studies are a more high-level overview of an area, charting possible sub-topics of interest to the research community, but without going to deep details, whereas typical surveys focus eactly on how the various methods have addressed research questions. The results of Mapping Studies can also reveal the trend in the scientific community regarding a particular area.

The methodology of compiling a mapping study is primarily based on research questions. High importance is also paid, of course, to the choice of research questions. Research questions should be clear, accurate and require specific information. All meta-data information is gathered as response to the demands of research questions. The next step, then, is to organize all this information, clear useless parts and keep only useful segments of information to reach to conclusions.

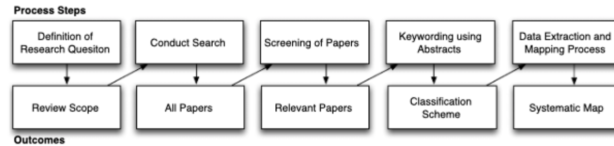

Figure 2.1: Mapping Studies Process [4]

.

Below we enlist the research questions used in our Study.

- RQ1) What is the annual amount of publications and the overall trend?
- RQ2) What is the breakdown of publications in terms of venues and types?
- RQ3) Which are the research questions and solution methods proposed in the literature for the area of schema evolution?
- RQ4) What kind of data(base) is the literature of schema evolution concerned with?
- RQ5) Does the paper concern on the evolution of the data and their schema, or does it also concern the evolution of the source code of surrounding application in sync with the schema?

The first two questions are *topic-independent* and can be applied to any surveyed research area. The two topic independent questions concern the ”when?” and ”where?” questions, and their breakdown can be used orthogonally to the following three, topic-dependent, research questions.

After defining the research questions, the next step concerns the search strategy. In other words, (i) where are we going to search for the requested information and (ii) which methodology we will follow in our search.

For our studies, validity of results was a major factor in the choice of source. So, for this reason, the rejection of Arxiv and Google scholar was necessary because these sources aren't peer-reviewed. Compared to the other sources, dblp is one of the bibliography repositories that contains material related with our subject. Thus, dblp was the tool that was used in the mapping studies for mining papers related with schema evolution.

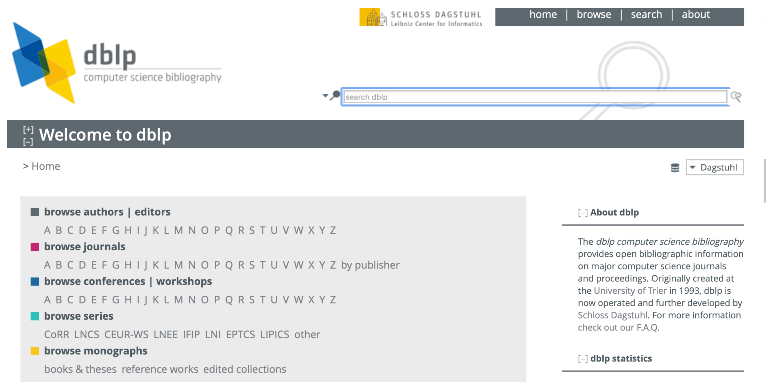

Figure 2.2: The homepage of dblp

Moreover dblp comes with a dedicated search facility `https://dblp.org/search/` that returns all the papers that it indexes, fulfilling a set of criteria for the title, venue and author names, as well as customization options.

It is remarkable to mention that our big problem wasn't in finding all information regarding schema evolution. The most difficult part of our study was how to extract and filter all information collected from dblp.

Results coming from dblp was in specific xml format. First step was to extract all this information without data loss. For this reason we developed a Python application that made very easily for us to extract only information needed in our study. Python offers a lot of advantages in this field of extraction, because many libraries are been available for extraction of data from xml files in an easy way.

After that we encapsulate in python application a filter controller that helped us a lot to organize and clarify this information. We apply some kind of inclusion and exclusion criteria so as to remove the useless publication that weren't needed in our study. Below is shown the list of inclusion and exclusion criteria that applied to our study.

Inclusion Criteria. We included papers fulfilling all of the following criteria:

1. publications concerning Schema Evolution

2. publications published in the aforementioned time-interval

3. include only accessible publications

4. choose only peer-reviewed publications

Exclusion Criteria. We excluded:

1. studies that were not authored in English
2. studies that have not been non-peer reviewed
3. studies whose full-text was not available to us (independently of reason, and including the request for purchase, the unavailability of an electronic version, or other reason)
4. non-original studies (e.g., multiple copies of the same publication),
5. studies were published before,
6. all other publications that are not matching inclusion criteria

After applying the criteria we had the final list of publications to examine and answer to the requested questions.

Count of Title of Paper	year										
Type	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	Total
Books and Theses											0
Conference and Workshop Papers	9	14	9	8	5	9	7	9	6	15	91
Informal Publications											0
Journal Articles	3	0	4	1	6	1	5	2	3	0	25
Parts in Books or Collections	2				1						3
Reference Works								1	1		2
Total	14	14	13	9	12	10	12	12	10	15	121

Table 2.1: Compilation of the corpus of papers.

The final step was to analyze these papers but didn't dive deeply. We cared about only to answer to the research questions we presented in the beginning. This was the purpose of our study and give quantity measurements.

For every paper in the final corpus, we didn't spend much time to analyze and give answer in each of request question. When we finished with all papers, we presented the results to diagrams and made the following conclusions.

Research Question & Answer

1. RQ1) *What is the annual amount of publications and the overall trend?*

   The rate of publications was pretty stable till 2019, with an average of 8 papers annually, but doubled in 2020 with 15 papers.

2. RQ2) *What is the breakdown of publications in terms of venues and types?*

   More that 87% of the papers were conference papers; also more than 85% were research papers, as opposed to other types, like demo's, position papers, etc

3. RQ3) *Which are the research questions and solution methods proposed in the literature for the area of schema evolution?*

   The study of how schema evolution takes place, how data are migrated, and how to automate evolution with consistency guarantees are the most popular research questions. The use of case studies for the understanding of how schema evolution takes place and methods and tools for the facilitating of the tasks related to schema evolution dominate the landscape in terms of methods used.

4. RQ4) What kind of data(base) is the literature of schema evolution concerned with?

   More than 64% of the papers were concerned with Relational database. Papers concerned with Non-Relational have minor percentage but there is a clear upward trend because there is a wide usage of this type of database in nowadays.

5. RQ5) Does the paper concern on the evolution of the data and their schema, or does it also concern the evolution of the source code of surrounding application in sync with the schema?

   More than 76% of the papers concerned with evolution in schema and data. A small percentage of papers is focus in evolution of schema queries and in evolution of source code in sync with evolution in schema.

Chapter 3

Problem Definition

3.1 What is a machine learning pipeline ?
3.2 What kind of stages can be in Machine learning pipeline?
3.3 Formal description and problem formulation
3.4 Framework used in development of machine learning pipeline
3.5 The MLlib library
3.6 Spark's API for Data management
3.7 How changes in external storage affect machine learning pipeline?

Our effort focuses on schema evolution in machine learning pipelines. To facilitate the discussion of the problem, first it is necessary to explain what a machine learning pipeline is and what kind of schema evolution can take place in the context of a machine learning pipeline.

3.1 What is a machine learning pipeline ?

A machine learning pipeline is a way to codify and automate the workflow it takes to produce a machine learning model. Machine learning pipelines consist of several steps in sequential order, with the ultimate goal to train a machine learning model.

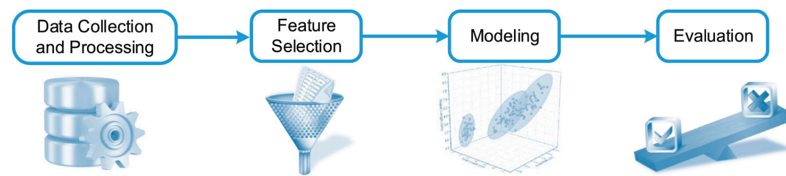

Figure 3.1: An example of a Machine Learning Pipeline[5]

A typical machine learning pipeline would consist of the following steps:

- Data Collection
- Data Cleaning
- Feature Extraction
- Building Model
- Model Evaluation

A Data Collector step involves the task of extracting data from a data source (a data file, a database, or other) and using them as inputs to the subsequent tasks. Next this data flows to the cleaning step. At this step removed duplicates data that might lead to less accurate predictions. After that is the process of transforming raw data into features that your pipeline will use to learn, that step is very important for algorithms that train a model to output a prediction or a classification. Until now these steps transform the data that will train the model. As data are ready next step develop the model while model evaluation find the best model that represents our data and how well the chosen model will work in the future.

3.2 What kind of stages can be in Machine learning pipeline?

A *Stage* is a fundamental unit of work in a pipeline. Alternative names in the workflow literature would be *activity* or *task*. Stages perform solid units of work, by taking a set of data records as inputs, performing a transformation or computation, and emitting a set of records as outputs.

There are two basic types of pipeline stages, specifically:

- Transformer
- Estimator

A *Transformer* is an task invoking an algorithm that transforms one dataset to another. The basic idea with Transformers is that they receive input in the form of structured records under a schema, which is enriched in the output with new columns containing the result of a computation.

To provide a concrete example of a Transformer, we can mention the *StopWordsRemover* transformer. In text processing, stop words are words which should be excluded from the input, typically because the words appear frequently and do not carry any particular meaning. A StopWordsRemover stage takes as input a sequence of strings and drops all the stop words from the input sequences, producing thus a sequence of strings as output that contain only words which are not "black-listed" as stop words. The list of stopwords can be specified by a stopWords parameter in the specification of a StopWordsRemover stage.

id	raw
0	$[I, saw, the, red, balloon]$
1	$[Mary, had, a, little, lamb]$

$\rightarrow$

id	raw	$filtered$
0	$[I, saw, the, red, balloon]$	$[saw, red, balloon]$
1	$[Mary, had, a, little, lamb]$	$[Mary, little, lamb]$

Figure 3.2: An example of a StopWordsRemover stage

In Figure 3.2 we give an example of a StopWordsRemover stage. As one can observe, the dataset has been changed. A new column called *filtered* has been added to the incoming dataset, and contains the subset of the input that does not contain stop words.

An *Estimator* stage is a task invoking an algorithm which can be applied to a dataset and produce a model that either fits the input data or trains over the input data. A very simple, but also quite commonly used Estimator is a *StringIndexer*.

Because most machine learning algorithms prefer to use numbers as input, if a dataset contains no numeric values, a StringIndexer can be used to convert these values to numbers. The example of Figure 3.3 shows how a StringIndexer Estimator

is working, by automatically assigning a numeric value to a text category (so, all a values are mapped to 0.0, b values to 2.0, etc).

id	$category$
0	a
1	b
2	c
3	a
4	a
5	c

$\rightarrow$

id	$category$	$categoryIndex$
0	a	0.0
1	b	2.0
2	c	1.0
3	a	0.0
4	a	0.0
5	c	1.0

Figure 3.3: An example of a Estimator stage

3.3 Formal description and problem formulation

In this section, we provide a formal definition of the entities involved in our problem.

Assume a set of simple data types $\mathcal{T}^D = \{T_1, \ldots, T_\Delta\}$ each with a name and domain of values, $dom(T)$, practically corresponding to integers, reals, strings, etc.

Assume a collection of *stage types* $\mathcal{T}^S = \{T_1, \ldots, T_\Omega\}$. Every stage type corresponds to a machine learning algorithm of a given library, that is to be instantiated by stages in the following. For the operation of the algorithm, a set of parameters needs to be fixed when invoked, so each stage type has a set of parameters $T.P$, which is a list of named elements, as pairs of names and simple data types $T.P = \{< P_1, T_1 >, \ldots, < P_n, T_n >\}$.

A *schema* is a list of named elements, as pairs of names and simple data types $S = \{< N_1, T_1 >, \ldots, < N_n, T_n >\}$.

A *stage* A is a tuple $A = \{T, I, O, P\}$ involving the following elements: T is a stage type, I, O are schemata and P is a list of named pairs of parameters $\{< P_1, v_1 >, \ldots, < P_n, v_n >\}$ (i.e., each parameter has a name and a specific value v belonging to the domain of the respective data type T, $dom(T)$). Thus, practically, a stage is the incarnation of a stage type with concrete input and output schemata and a concrete specification of the parameters.

A *data set* is a trivial case of a stage with T: $Dataset$, $I = O =$ a certain schema S, P: $\emptyset$. Thus data providers and data sinks were data reside and are eventually stored,

respectively, can be treated as trivial stages.

A *general-case pipeline* $\mathcal{P}$ is an acyclic directed graph $(\mathcal{V}, \mathcal{E})$, with stages as the nodes of the graph and directed edges between the stages. Every edge $< s, t >$ carries a mapping M: $s.O \rightarrow t.I$ mapping the output of the previous stage to the input of the next. The fountain nodes of the graph are obligatorily data sets.

A *linear pipeline* $\mathcal{P}$, or simply *pipeline*, is a *list* of stages, implementing a line graph. Implicit in the definition is the mapping of the output of a stage A_i to the input of the stage A_{i+1}. The first stage of the linear pipeline must be a data set.

In the rest of our deliberations, unless otherwise specified, the term 'pipeline' refers to linear pipelines.

Problem formulation. This work addresses two fundamental problems:

1. Assuming a collection of source files $\mathcal{S}$ containing arbitrary statements in a programming language that, among other define a pipeline, extract an abstract representation of the pipeline $\mathcal{P}$ out of $\mathcal{S}$

2. Assuming that the schema of a data set participating in a pipeline $\mathcal{P}$ changes, identify all the stages of the pipeline that are affected as a collection of affected stages $\mathcal{P}^A$, $\mathcal{P}^A \subseteq \mathcal{P}$

3.4 Framework used in development of machine learning pipeline

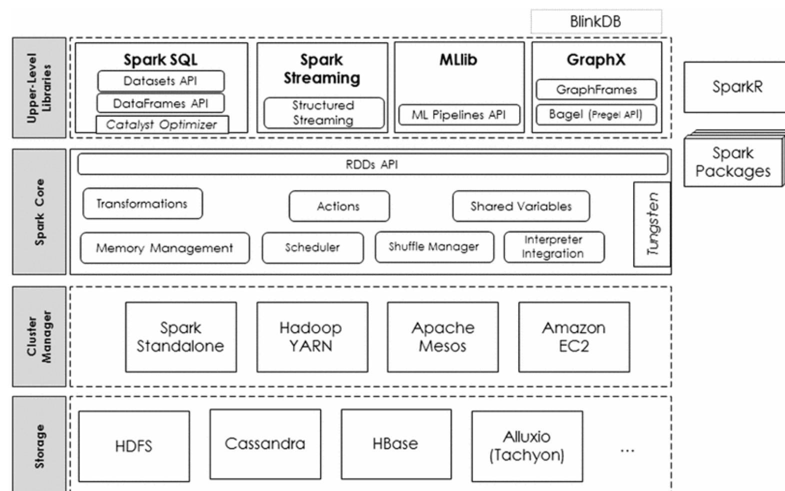

Figure 3.4: Apache Spark Architecture[6]

In the development of our machine learning pipelines we have chosen Apache Spark[6]. Apache Spark is a framework used in common especially for big data and data analytics.Apache Spark was initially developed in University of Berkeley as a research project in 2009. The core motivation around Spark was the idea of providing an environment that could (a) provide programming libraries for the most common data engineering tasks (specifically, SQL-like querying for stored data, machine learning algorithms, graph management algorithms and stream management), and, (b) operate on a massively distributed platform, in a manner transparent to the programmer.

In Figure 3.4 we depict the fundamental architecture of Spark. Apache Spark uses many cores to split the jobs. This functionality enables to develop large-scale machine learning algorithms that can support large volumes of data to be processed in a parallel way, in order to be more effective.

Apache Spark is a powerful big data processing platform which adapts the hybrid framework. A hybrid framework offers support for both batch and stream processing capabilities. Even though Spark uses many similar principles to Hadoops MapReduce engine, Spark outperform the latter in terms of performance. For instance, given the same batch processing workload, Spark can be faster than MapReduce due to the "full in-memory computation" feature used by Spark compared to the traditional read from and write to the disk used by MapReduce. Spark can run in standalone mode or it can be combined with Hadoop to replace MapReduce engine[7].

A distinctive characteristic of Apache Spark is its ability to support various languages such as Python, Java and R in an easy-to-use fashion. Programs written on top of the Apache Spark framework are regular programs, e.g., in Java, that can use the data types provided by Spark and invoke the methods that they support, thus facilitating the incorporation of all the aforementioned functionalities of Spark into an otherwise typical program in one of these languages[7].

Taking all the above reasons into consideration we conclude that the Apache Spark is the most appropriate tool for the development of machine learning pipelines.

3.5 The MLlib library

Of particular interest to our deliberations is the fact that Apache Spark has embedded a library, called *MLlib*, specifically targeted towards providing a collection of machine learning algorithms that can be pro grammatically invoked.

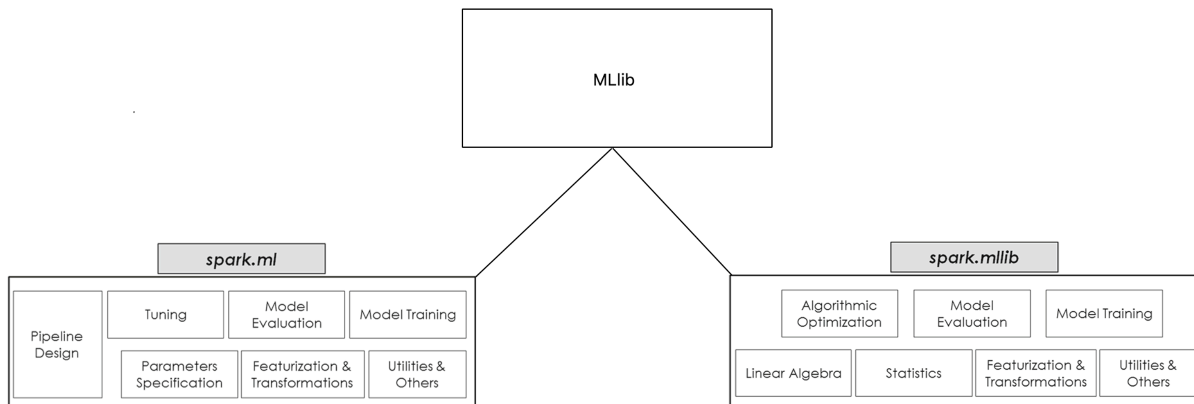

Figure 3.5: MLlib library [6]

MLib is Spark's open source distriuted machine learning library. MLlib provides a lot of statistical, optimazation and linear algebra primitives. With the Spark, supports a variety of languages. Also it contains a high-level api that is called Pipeline API that simplify the development of end-to-end machine learning pipelines to Spark ecosystem.[8]

3.6 Spark's API for Data management

Apache Spark use three different API's for working with Data.

- **RDDs**
  RDDs or Resilient Distributed Datasets was the first and fundamental data structure of Spark, introduced in 2011. Data of this structure can be stored in different nodes of cluster that allows to be processed in parallel in a location-transparent and failure-resilient way.
- **Dataframes**
  Dataframes is the next data structure introduced in Apache Spark after RDD, introduced in 2013. Dataframe is a two-dimensional array with columns and rows, much like a table in relational database.

- **Datasets**

  Datasets is the last data structure introduced in Apache Spark, introduced in 2015. It is an extension of the Dataframe data type that encapsulates benefits from other previous data structures.

A fundamental difference between RDDs and Dataframes is that the former are simply collections of objects whereas the latter allow named columns in their definition – very much simulating a relational table. RDD's, being collections of objects are necessarily typed; thus, they allow the invocation of methods like filter, map to subgroups, saving, etc with respect to characteristics of the Class of the contents of the RDD. On the other hand, Dataframes are not exactly typed in terms of the host programming language, but rather they are collections of the data type Row. A Row has indeed columns specified, and then, a Dataframe can allow the invocation of methods that resemble parts of an SQL statement (e.g., select(), filter(), groupBy(), with column names as parameters of the operations).

A Dataset is an evolution of both Dataframes and RDDs to bring the benefits of both worlds. A Dataset is typed with respect to the host language, i.e., it is defined as a collection of objects of a class of the host program. At the same time, it also provides a way to be treated as a Dataframe (practically, a Dataframe is a $Dataset < Row >$), thus allowing the SQL-like querying of the data.

3.7 How changes in external storage affect machine learning pipeline?

In most cases, a machine learning pipeline will be loading the data from external storage into datasets. But, as we mentioned before, a machine learning pipeline consists of many stages that refer to columns of a dataset schema. The specific schema is strongly connected with external datasource.

So, every time the schema of the external data source is changed, this immediately affect the syntactic correctness and, consequently, the functionality of the machine learning pipeline. The cause of problem is that there are dependencies between the stages of pipeline and the external data storage.

To illustrate the issue, let us assume the following scenario: We have a simple machine learning pipeline that consists of 2 stages:

The external data source is a csv file that contains 4 columns: { id, featurecol1,

	Stage	*Input Attributes*	*Output Attributes*
1	VectorAssembler	{ featurecol1, featurecol2, featurecol3 }	{ features }
2	MinMaxScaler	{ features }	{ sfeatures }

Table 3.1: An example of dependency to schema attributes

featurecol2, featurecol3 }. Then, data prorgessively pass from a vector assembler and a min max scaler.

It is obvious that there is dependency between the stages of the pipeline and the external data source in direct way. If, for some reason, column names were changed or columns removed from the csv file, the machine learning pipeline will simply fail to run. In a nutshell, the main problem is that when the schema of the dataset changes, this affects the stages of the pipeline and eventually the machine learning pipeline fails.

Chapter 4

Pipeline Abstraction and Evolution

4.1 Abstract Syntax Trees

4.2 Eclipse JDT

4.3 What is the Java Model of Eclipse JDT?

4.4 Eclipse Plugin Projects

4.5 How to find dependencies in ML Pipeline?

4.6 Correctness requirements and Spark particularities

4.7 Modeling of the information extracted from ASTParser via graph diagrams

4.8 How schema changes can be detected

The aim of this Thesis is to provide mechanisms that will facilitate (a) given a pipeline that is programmatically defined within a source file in an ad hoc manner, to abstract it in a principled manner, and (b) to calculate the impact of the evolution of the source file schema into the elements of a pipeline, based on the aforementioned abstraction.

As we have already seen from the previous section, the main cause of concern is the existence of dependencies between the dataframes and the stages of pipeline. Therefore, we need to statically analyse the source code and extract the stages and dependencies within the pipeline. A powerful tool for the static analysis of source code, is the AST Parser. Thus, in the next subsection we will explain what is the AST Parser.

4.1 Abstract Syntax Trees

An Abstract Syntax Tree (AST) is produced during the static, syntactic analysis of source code by the compiler and represents, in a tree structure, all the information about the structure of the source code. This hierarchical, abstract representation is then used from the compiler for the next stages of compilation, in order to produce the final executable. An AST Parser (short for Abstract Syntax Tree Parser) is a software module that parses the source code in order to produce its respective Abstract Syntax Tree.

Abstract syntax trees are structures in a compiler's intermediate phase. They serve as interfaces between the parser and the compiler's later stages. All information about a source program's syntactic structure is stored in abstract syntax trees, but there is no semantic interpretation. These trees can be walked to seek for primitive or derived smell features in the source code. The ASTVisitor class in Eclipse comes with a ready-to-use utility for traversing these trees.[9]

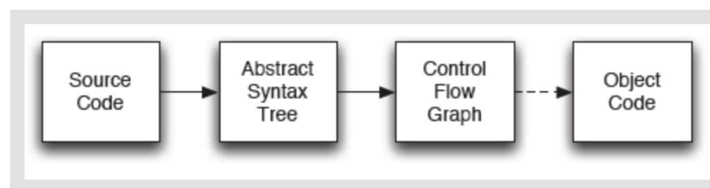

Figure 4.1: Stages of Compiler.

In the example of Figure 4.2, we see how a small loop presents a hierarchical structure, starting with a while node at the root of the respective tree, being ”detailed” into (a) a condition and (b) a body, which in turn are recursively ”detailed” by their constituents: programming statements, variables, conditions and values.

As we see from Fig 4.2 abstract syntax tree is comprised of AST Nodes. Each AST Node represent a particular construct in the concrete syntax. For this reason, before construction of Abstract Syntax Tree, the defining of syntax rules is necessary to come first, because based on these rules compiler build the Abstract Syntax Tree.[10]

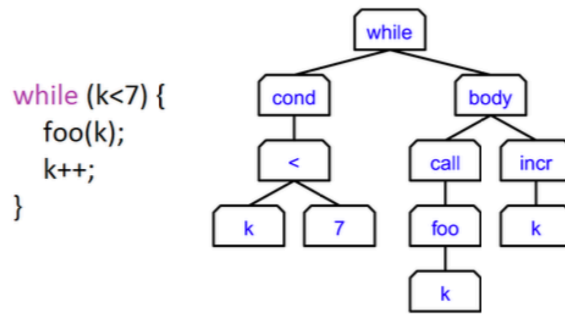

Figure 4.2: Abstract Syntax Tree Example

4.2 Eclipse JDT

The machine learning pipelines that we handle have been written in Java. In our project we use Eclipse as our Integrated Development Environment (IDE) because it offers many benefits regarding the choice to use Java language to develop the machine learning pipelines. The collection of Eclipse Development Tools (JDT) provides an API with classes and methods related to Abstract Syntax Trees that allows to automatically extract the AST of an entire project.

1. Java Model
2. Abstract Syntax Tree

In this context, JDT parsing has every Java source file to be represented as tree of AST nodes. Each node is a subclass of ASTNode. Each subclass is related with specific element of Java language. There are nodes for method declaration ('MethodDeclaration') or nodes for VariableDeclarations('VariableDeclarationFragment') of for other assignments and so on.[11]

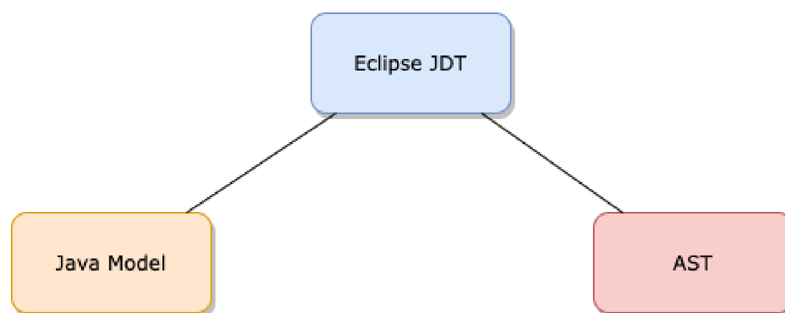

Figure 4.3: Eclipse JDT.

4.3 What is the Java Model of Eclipse JDT?

In the context of Eclipse JDT, each Java project is internally represented via a *model*. The difference between an AST and a model is as follows: as we have already said, an Abstract Syntax Tree (AST) keeps all the information related with source code; at the same time, all the information about the structure of a project (folders, packages, modules, classes) is kept in the Java Model.[11]

Static analysis, in accordance with the hierarchical nature of ASTs and the Java Model, proceeds in a top-down fashion, in order to extract the information of the project structure and the source code and to abstractly represent it in a uniform way.

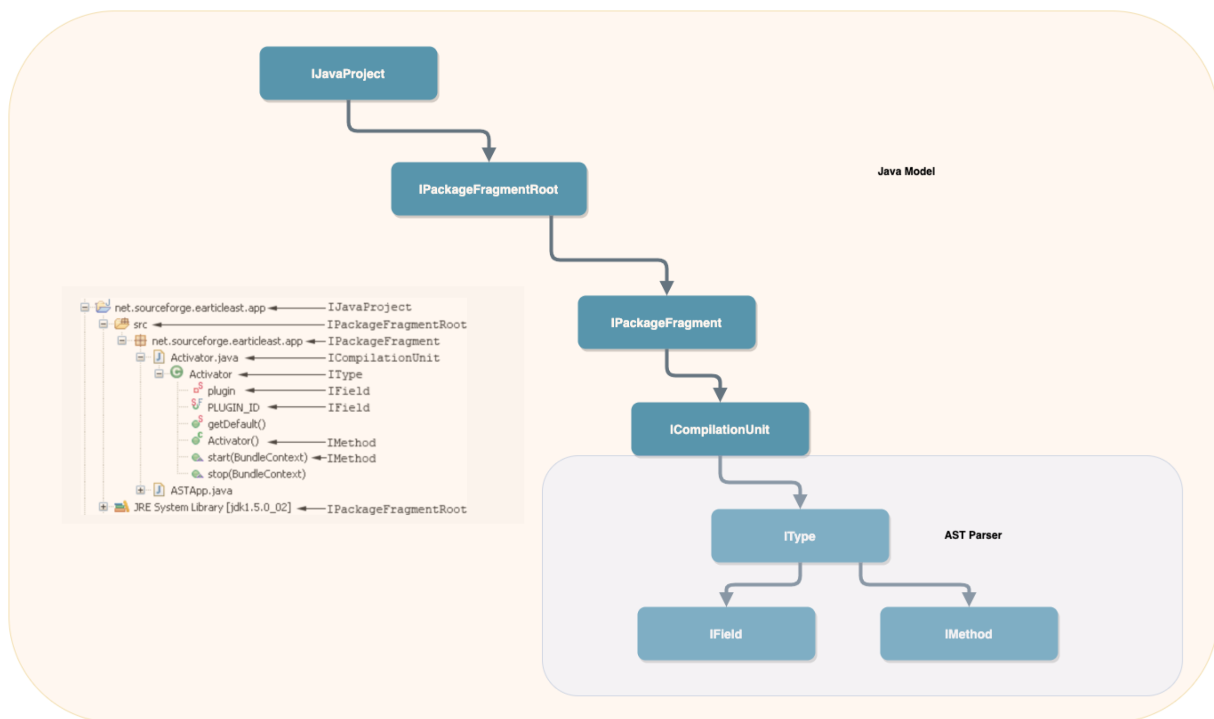

Figure 4.4: JavaModel Structure

In Figure 4.4, initially, we use the Java Model to get the java project we want to analyze. The next step is to drill-down into the structure of the project and find the packages of project. Inside the packages we focus only in the source packages of project. Within these source packages, we can look for the Java classes that we care about, and which contain the programmatic definition of our pipelines. As soon as we get at the level of classes and their corresponding source files, we use the AST Parser that will help us abstract their contents and identify the pipelines and their elements.

4.4 Eclipse Plugin Projects

Eclipse provides the possibility of defining and using Plugin Projects. A Plugin Project can be executed like a typical Eclipse application and then, it creates a runtime application where we can load our projects and have them analyzed with respect to the Java Model. Thus, the Java Model can be used from a Plugin Project in order to get all the information from the workspace where several projects have been loaded.[12]

Given the above capabilities, the first step of our implementation is to build a plugin project via Eclipse IDE. The next step is to choose the project we want to analyze. We check if it is Java project and convert it to Java Core project for analysis. After that we keep only the source packages included in project (as it is possible that packages of other purposes can be part of a Java project too). For every source package we get all the classes included in it. Java Model represents every class as an instance of *ICompilationUnit*. Until this depth of analysis, the only API used was the Java Model because it is the Java Model that is responsible for extracting the structure of project.

Then, the AST Parser is invoked for the static analysis of *ICompilationUnit*. After this level, and inside the components of an *ICompilationUnit* we can find only source code.

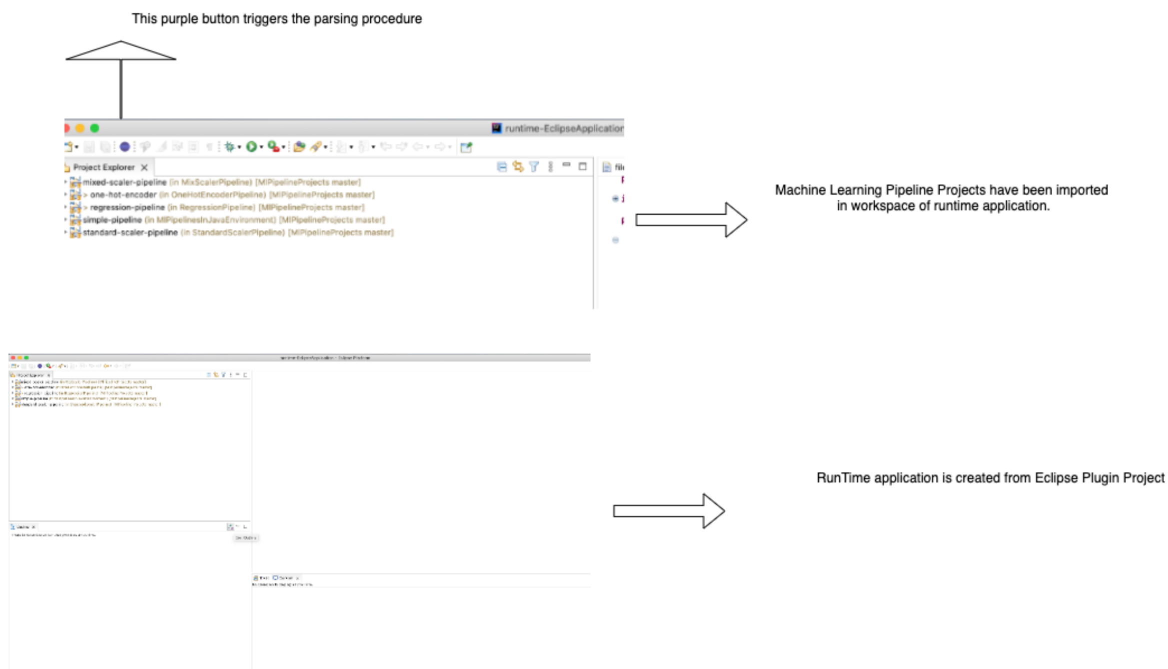

Figure 4.5: Eclipse Runtime Application.

4.5 How to find dependencies in ML Pipeline?

Now, we are ready to statically analyse a class, which is the main element of source code.

In every class, there is a plethora of declarations such as import declarations, variable declarations etc. We focus specifically on method declarations because it is exactly inside methods where the machine learning pipelines are defined.

The next step is to analyze the statements inside a method declaration. A machine learning pipeline statement has a specific definition used inside a Java Spark Application; thus, we have to identify the statements that pertain to this definition in order to find the machine learning pipeline statement. The typical format of a pipeline definition is the creation of a *Pipeline* object and the assignment of stages to it (see Figure 4.6).

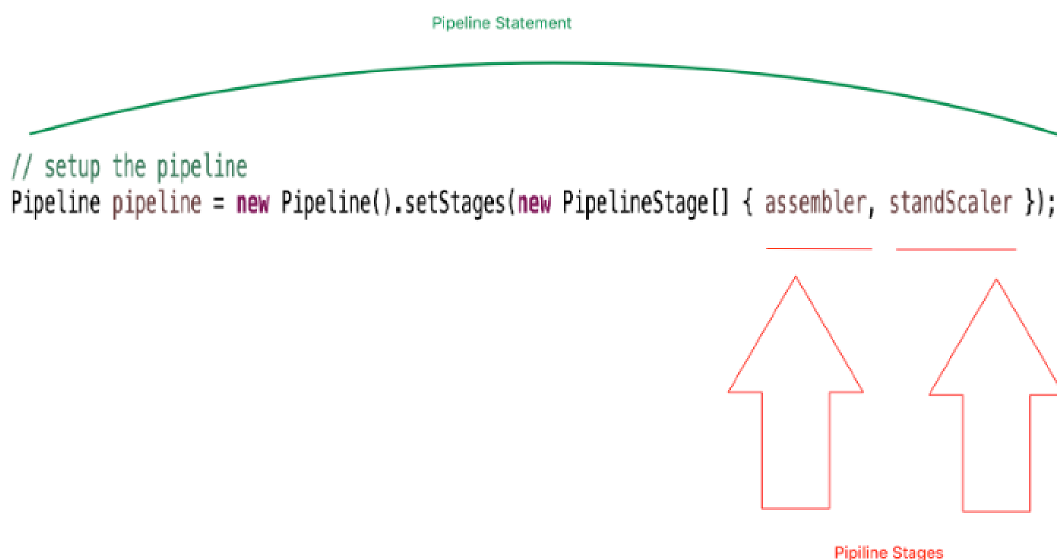

Figure 4.6: Pipeline Statement.

As Figure 4.6 demonstrates, the stages of the pipeline are used as arguments in the pipeline declaration. Getting this information about the number of stages of the pipeline and their names, we then can proceed to the next step, which is to find the statements that define the pipeline stages, inside the method declaration. Specifically, from the pipeline declaration, one can obtain the information about the names of stages participating in the pipeline. So, the next action is to look in variable declarations of methods and lookup the name of variable. If the variable name is equal with the stage defined in pipeline and the type of the variable belongs to common

pipeline stages, as provided by the specification of Spark ML, the variable statement is recognized as one of the stages of pipeline.

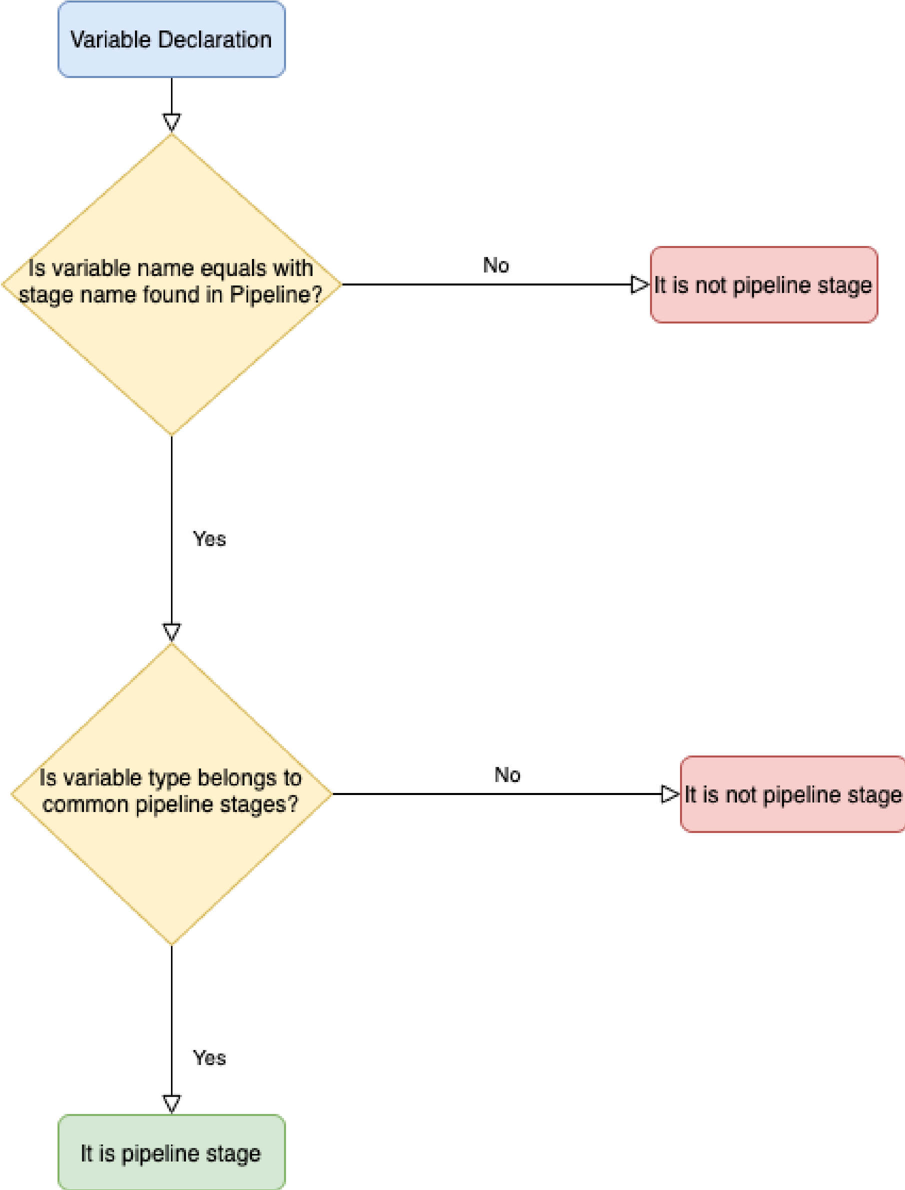

Figure 4.7: Flow diagram recognize Pipeline Stage.

Each pipeline stage contains as input arguments columns coming from (a) either a dataframe of external source or (b) a previous pipeline stage, emitting them as its output. Also, a stage has output arguments for columns that are produced during this stage and used as input in the next stages of pipeline. Keywords (”setInputCol”, ”setInputCols”) make easy to recognize input arguments. In the same way, keywords (”setOutputCol”,”setOutputCols”) can be used to define output arguments. Based on these characteristics, we can find in the stage statement input and output arguments.

Moreover some types of stages such as *LogisticRegression* or *DecisionTreeRegressor* have different keywords in their statement that define outputs or inputs ("setFeaturesCol", "setLabelCol"). The ASTParser that we have implemented also recognize these types and collects inputs and outputs of these stages. Figure 4.8 gives an example of the aspects related to the specification of a typical stage and Figure 4.9 an example of non-typical, extended set of keywords.

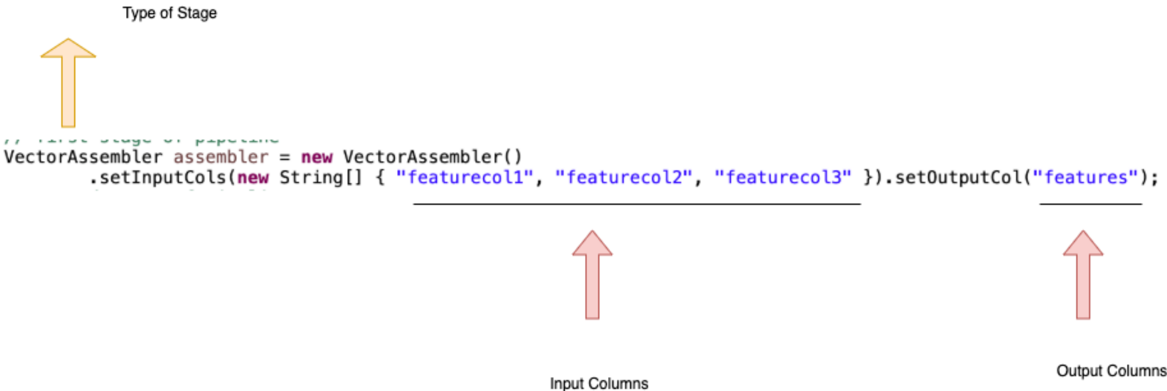

Figure 4.8: Pipeline Stage Statement.

```
DecisionTreeRegressor regressionModel = new DecisionTreeRegressor().setLabelCol("labelIndex")
        .setFeaturesCol("features");
```

Figure 4.9: Pipeline Stage Statement with non-typical keywords.

All the information about pipeline stages that have been extracted via AST Parser is stored in a text file, for subsequent exploitation by tools like the *GraphAnalyzer* that will be detailed in the sequel. A FileExporter stores this information to a specific file called `stages.txt` in specific format. The exported file contains, for each pipeline found in the project, (i) the stages of pipeline, (ii) the input columns of each stage, and, (iii) the output columns of each stage. Moreover, the exported file contains the dataframe columns that come from external source. In our Study we suggest that all pipelines get the data from only one dataframe. All pipelines are being supplied by one external source. A typical example of export file is shown in Figure 4.10.

```
columnsOfFile:[id, featurecol1, featurecol2, featurecol3, feature_1, feature_2, feature_3, feature_4,
          label, text, labelDesicion, feature_dis1, feature_dis2, feature_dis3, category_1, category_2]
-------
------start of pipeline0-----
node1: StringIndexer
inputs for node1:["category_1", "category_2"]
outputs for node1:["category_1_index", "category_2_index"]
-------
node2: OneHotEncoder
inputs for node2:["category_2_index"]
outputs for node2:["category_2_OHE"]
-------
------end of pipeline0-----
------start of pipeline1-----
node1: StringIndexer
inputs for node1:["feature_2", "feature_3"]
outputs for node1:["feature_2_index", "feature_3_index"]
-------
node2: OneHotEncoder
inputs for node2:["feature_2_index", "feature_3_index"]
outputs for node2:["feature_2_encoded", "feature_3_encoded"]
-------
node3: VectorAssembler
inputs for node3:["feature_1", "feature_2_encoded", "feature_3_encoded", "feature_4"]
outputs for node3:["features"]
-------
node4: LogisticRegression
inputs for node4:["features"]
outputs for node4:["label"]
-------
------end of pipeline1-----
------start of pipeline2-----
node1: VectorAssembler
inputs for node1:["featurecol1", "featurecol2", "featurecol3"]
outputs for node1:["features"]
-------
node2: MinMaxScaler
inputs for node2:["features"]
outputs for node2:["sfeatures"]
-------
------end of pipeline2-----
```

Figure 4.10: An example of file exported by ASTParser Application

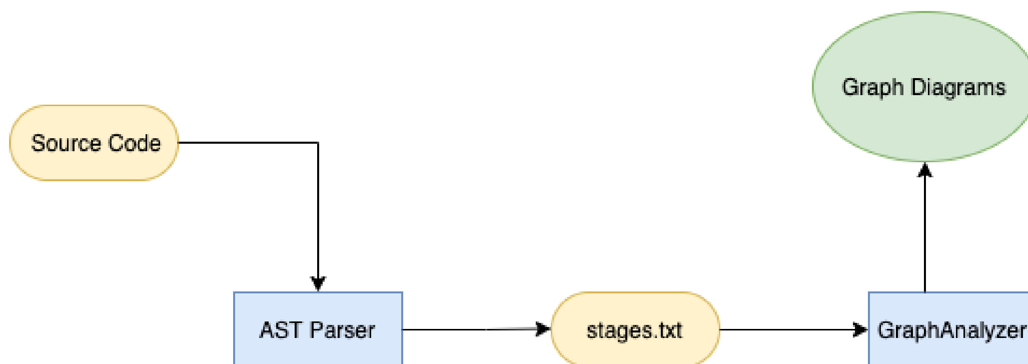

Figure 4.11: Architecture of Application

4.6 Correctness requirements and Spark particularities

In this section, we discuss some particularities that pertain to Spark and the requirements for the source code of the scanned projects, such that our tools work correctly.

First, note that, in terms of Spark terminology, Transformer stages have input and output columns, with input columns being the parameters that are employed in the algorithm that the stage executes and output columns being the attributes generated by Spark. Typically, the data produced from a transformer retain their original schema, enhanced with the new output columns, Estimators on the other hand, instead of input and output columns might have feature columns that regulate which attributes will be used for the model fitting and label columns as the output columns with the result. We uniformly refer to them as input and output columns, and respectively, schemata.

Second, we would like to clarify the term input and output schemata. In Spark, the structure of the data that are emitted from a stage typically contains (a) the original columns, and, (b) the new columns that are generated. Progressively, as data pass through the stages of the pipeline, the attributes of the data become more and more with the additions of the new attributes. However, when Spark refers to *input schema* of a stage, it does not refer to the structure of the incoming data (that can be arbitrary as far as the compiler knows) but to the binding of the parameters of the stage's algorithm to specific attributes that must exist in the structure of the incoming data.

For the method to work properly, we require the following constraints to be respected in the configuration of the pipelines of the project that we work with.

1. The method works with Apache MLib pipelines: this means that, programmatically, the source code refers to the data types and semantics of the pipelines of Spark. This implies that the pipeline is linearly executed, and that, at execution time, the output schema of a stage is the union of the output schema of its previous stage with the generated attributes at the output schema. This is very specific for Spark, and allows the arbitrary placement of stages in a linear pipeline.

2. The project is based on a single data set which feeds all the pipelines

3. Each attribute name is generated (either as part of the data frame schema, or as generated attribute from a stage) only once (a violation of this can also lead to execution errors in Spark)

4. A stage is well-defined if (a) its input schema is programmatically specified (i.e.,

the attributes that will act as parameters for the stage are specified in the source code), and, (b) the output schema is also programmatically specified (i.e., all generated attributes are named). Note that if this is violated, (i) Spark crashes at runtime if the input columns are not specified, and, (ii) automatically generates a default name for the output of the stage, if a name has not been set. However, using such an automatically name later in the pipeline is a rather precarious tactic. If the attribute is not used at all, then there is no problem with not dealing with it anyway. Thus, we can anticipate that a well-defined pipeline, will have all the input and output parameters specified and named.

4.7 Modeling of the information extracted from ASTParser via graph diagrams

All the information about the dependencies between the stages of each pipeline and the external sources (dataframe) is contained inside the `stages.txt` file. To represent visually this information as well as to detect the impact of changes, we have developed a new Maven Java Project called *GraphAnalyzer*.

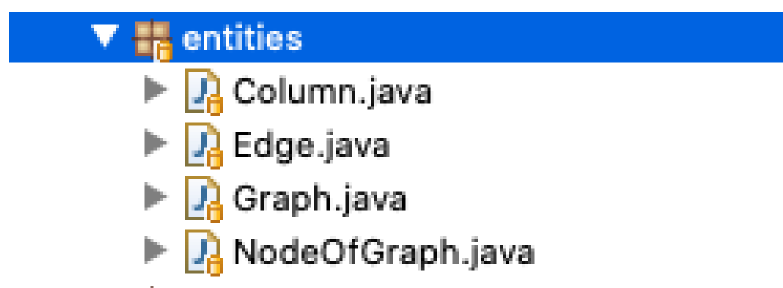

Figure 4.12: Entities of GraphAnalyzer

An important functionality of *GraphAnalyzer* is to visualize (via the facilities of the JavaFX library) the pipelines as graph diagrams. The tool parses the `stages.txt` file, and, for each stage or dataframe reported there, an instance of *NodeOfGraph* is created. Furthermore, the pipeline name is also stored inside the *NodeOfGraph* instance such that we can always know in which pipeline, the node belongs to. Also, information related with input and outputs fields are kept inside of *NodeOfGraph* that are necessary in the design of edges between the nodes. The first step is completed by a *FileReader* object and, at the end of the step, we have already gathered all the

nodes participated in graph.

4.7.1 The Name-Dependency Graph

The most critical part of the application, is to be able to trace the different stages of the pipeline are dependent upon previous parts of the pipeline.

Definition [Name-Dependency Graph]. Assuming a set of attribute names $\mathcal{A}$, the $Name - Dependency\ Graph$ is a labeled graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, with a labeling function $\mathcal{E} \rightarrow 2^{\mathcal{A}}$ such that:

- the set $\mathcal{V}$ includes all the stages of the pipeline along with the dataframe
- we say that node t *name-depends* on node s, $nameDepends(t, s \mid a)$, and there is a directed $< s, t >$ edge between a node s and a node t if and only if there is an attribute a in $t.inputSchema$ that is generated for the first time in $s.outputSchema$
- the label of every edge $< s, t >$ is the set of attribute names a_i for which we have $nameDepends(t, s \mid a_i)$

We assume that dataframes have inputSchema equal to their outputSchema.

So, practically, all this boils down to the following simple graph:

- all the stages and the data set form the nodes of the graph;
- edges stand for the fact that the source of the edge is the first place where we encounter an attribute that appears in the input schema of the target

What the latter statement means is based on the fundamental idea that when an attribute is specified as part of the input schema of a stage, this means it is explicitly selected to serve a role as a parameter for the function the stage performs. The stage is syntactically erroneous if this attribute is removed from the graph, so, we need to trace where the attribute is generated for the first time. Thus, *an edge signifies a dependency relation between the node that generates an attribute and a node that, later, uses this attribute as a parameter for its algorithm*. Equivalently, the edges signify the answer to the question: *assuming a node disappears from the graph, or, an attribute is removed from the output schema of a node, which nodes are immediately affected?*

The facilitator of the name-dependency graph creation is the service called $GraphCreator$. This service is responsible for handling the edges of the graph. Edges represent the

dependencies between nodes i.e., the stages of pipeline. As there is no dependency between nodes of different pipeline, the logic of GraphAnalyzer is to add edges between nodes belonging to the same pipeline. So first stage is to add edges only to nodes belongs to the same pipeline.

To find the edges between nodes of the same pipeline, we use the information of input,output fields contained in each node. If the output field of nodeA is equal with the input field of nodeB, it means that there is dependency between nodeA and nodeB, so add an edge between nodeA, nodeB.

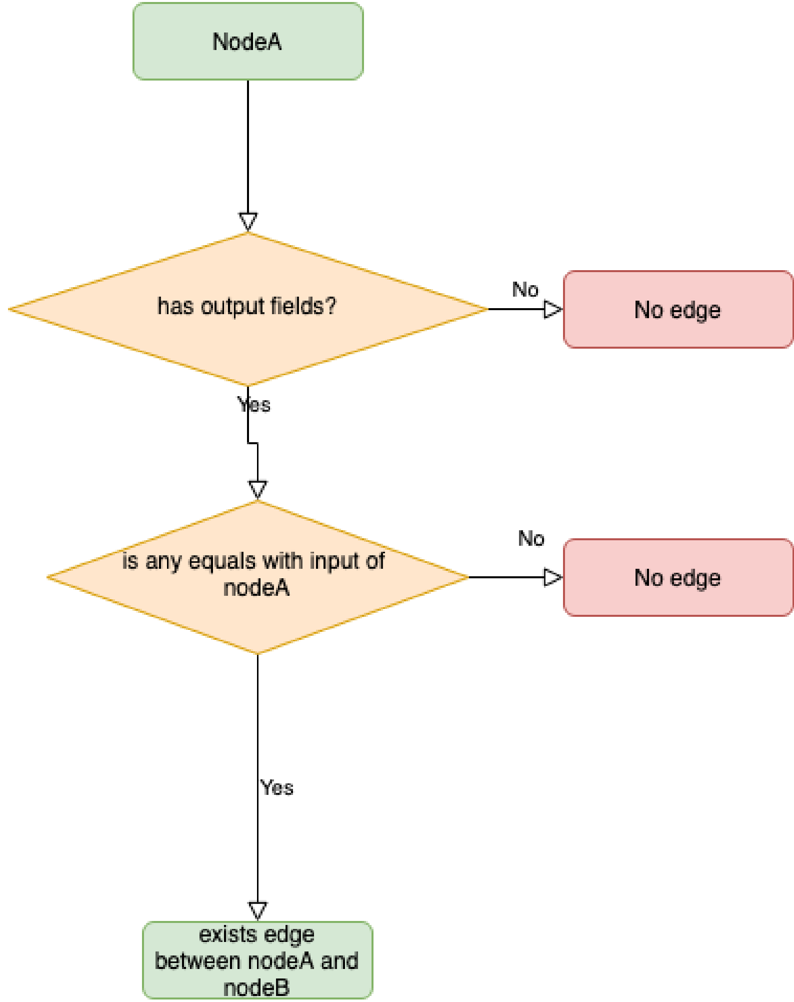

Figure 4.13: Flow diagram which show how to recognize edge between nodes.

After adding all edges between nodes of the same pipeline, we are missing only the edges coming from a external sources (dataframe). Observe Figure 4.14. The figure depicts a case with 5 different pipelines. From the figure we can conclude that what is missing in all pipelines is the tracing of the external dependencies, i.e, dependencies coming from external datasources (dataframes).

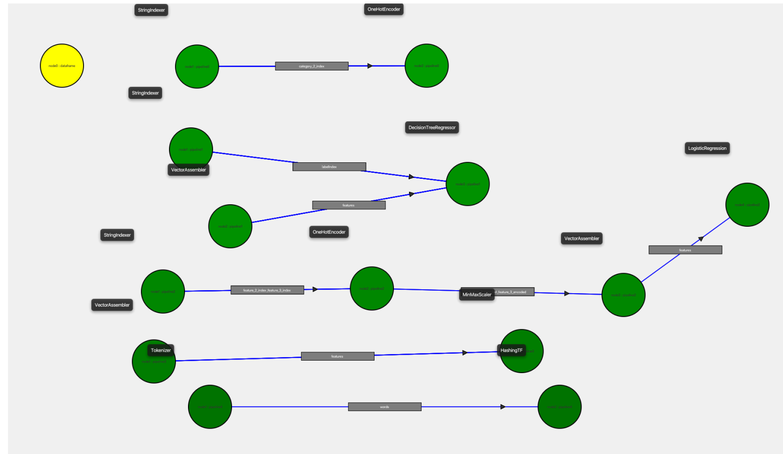

Figure 4.14: Graph with no external dependencies.

The next step is, then, to add edges coming from external source (dataframe). Looking through every node in graph, we check the input fields of every node. If the input field doesn't come from another node in a pipeline, this means that comes from external source.

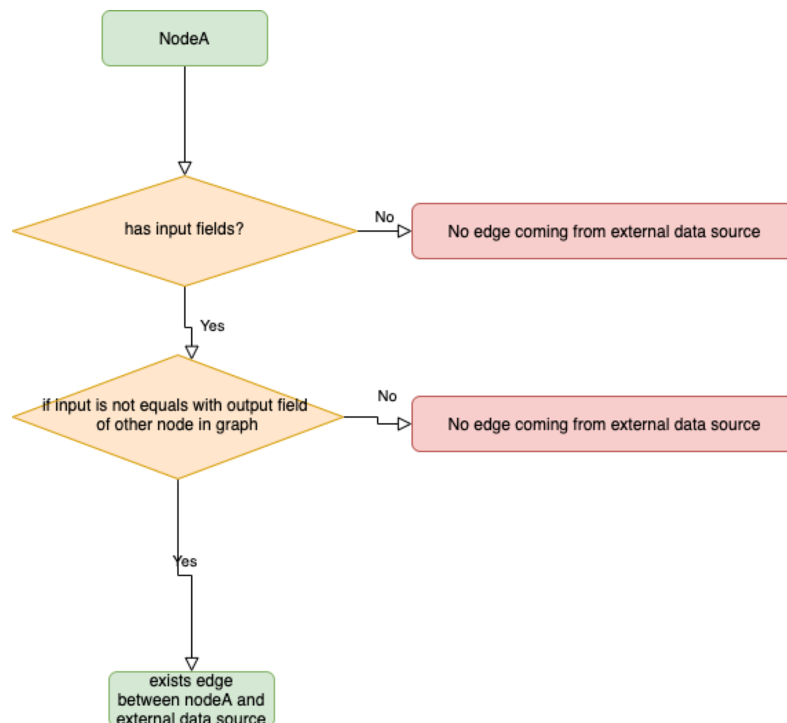

Figure 4.15: Flow diagram which show how to recognize edges comes from external data source

After recognizing which nodes of graph have direct connections with external data source, we add the edges coming from the dataframe in the graph.

In terms of visualization, we create a graph with the nodes of the different pipelines. A different color is used to in the designing of these edges. Red color has been chosen for the designing of edges coming from dataframe. Coming back to the example of Figure 4.14, the final graph after adding all the edges is visualized in Figure 4.16.

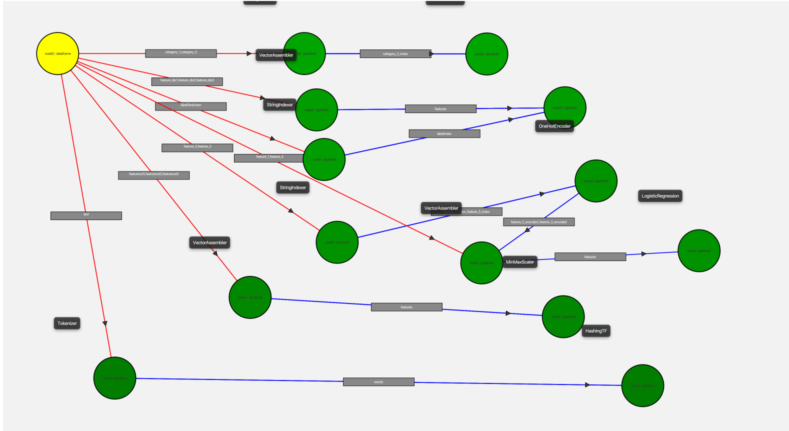

Figure 4.16: Graph with external dependencies.

4.7.2 Topological sorting

After the graph has been constructed and designed, we apply the algorithm of topological sorting in the graph. The result of this action is to reorder the nodes of the graph and to check if there is a cyclic dependency to the graph. The pseudocode of the algorithm is shown below. Before applying the algorithm to the graph, we firstly

save all the edges of the graph because are going to be deleted during execution of algorithm. After execution, nodes of graph have been reordered and we add again the edges we have saved before.

Algorithm 4.1 Topological Sorting of Graph

```
1: L ← Empty list that will contain the sorted elements
2: S ← Set of all nodes with no incoming edge
3: while S is not empty do do
4:    remove a node from S
5:    add n to L
6:    for all node m with an edge e from n to m do
7:       remove edge e from the graph
8:       if m has no other incoming edges then
9:          insert m into S
10:      end if
11:   end for
12: end while
13: if graph has edges then
14:    return error
15: else
16:    return L
17: end if
```

Main purpose of topological sorting is to detect cyclic dependency if exists and prevent from designing the graph to JPanel when exists a cyclic dependency.

4.8 How schema changes can be detected

Change in any column of the dataframe of a pipeline can cause problems to the machine learning pipeline, if this column has a direct dependency with any stage of pipeline. For this reason, we have built the *GraphAnalyzer* in a way that it can handle these changes and report them to the analyst.

The *GraphAnalyzer* can focus on external dependencies coming from external datasource. There is a check in the GraphAnalyzer between external dependencies coming from external datasource and the expected inputs of machine learning pipeline. This way, as we see in the example of Figure 4.17, the system can detect missing dependencies. Machine learning pipeline execution will fail because of

missing dependencies.

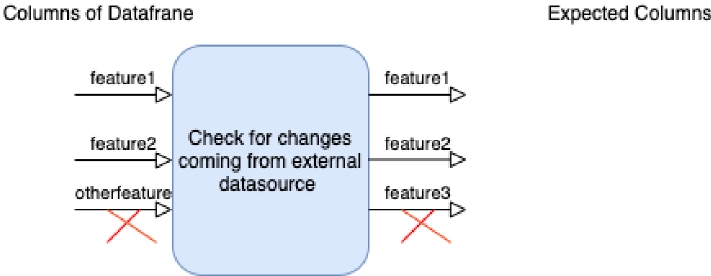

Figure 4.17: System checks for missing dependencies .

The next step of the system is to find out which pipeline uses the specific missing dependencies. Moreover, it will find out which steps of the pipeline have direct relation with missing dependencies. Thus, the affected nodes are detected.

In terms of visualization, the system will color red all the nodes - steps of pipeline that have a problem with missing dependencies. Also, in the weight of every edge, there is information about the input column of each node - stage. The missing dependency will be colored also red to inform which column is missing and causes problem to pipeline.

To explain the situation better, let us present the following example with its state (a) before and (b) after schema changes. There are 5 different pipelines under the project. All these pipelines are being sourced by the same external dataframe.

Before any schema change there is no missing dependency so, all the nodes have the expected green color and all pipelines are running without problem (Figure 4.18).

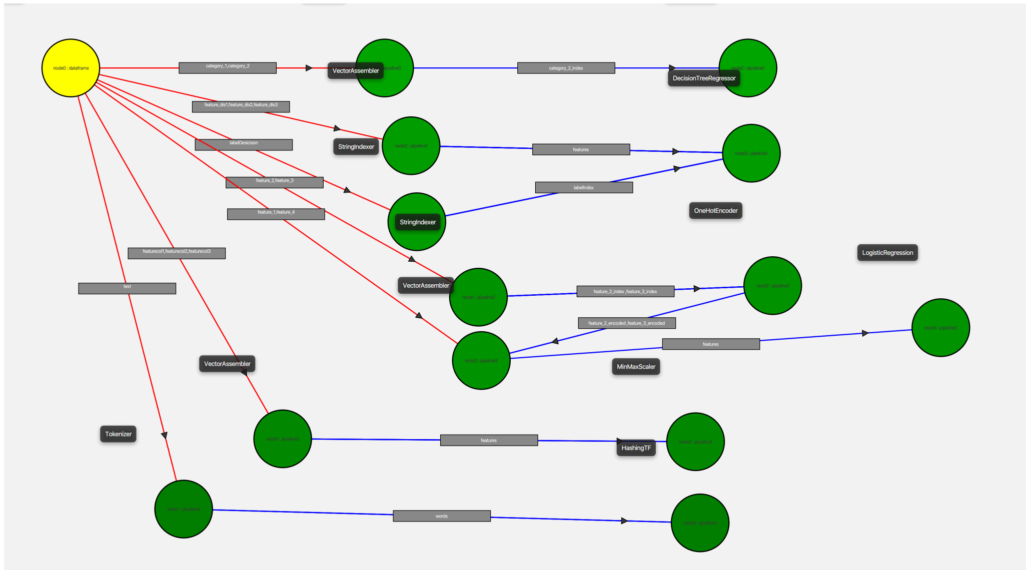

Figure 4.18: Graph before schema change

When two columns of a dataframe (specifically: $featurecol2$, $labelDesicison$) are altered, this affects the pipelines $pipeline1$, $pipeline3$, so we expect in the final graph these pipelines to be colored red. As expected, in Figure 4.19 all nodes of pipeline1 and pipeline3 have been colored red. Also, the columns that are missing have the red color (although not easily discernible in the figure, due to space constraints).

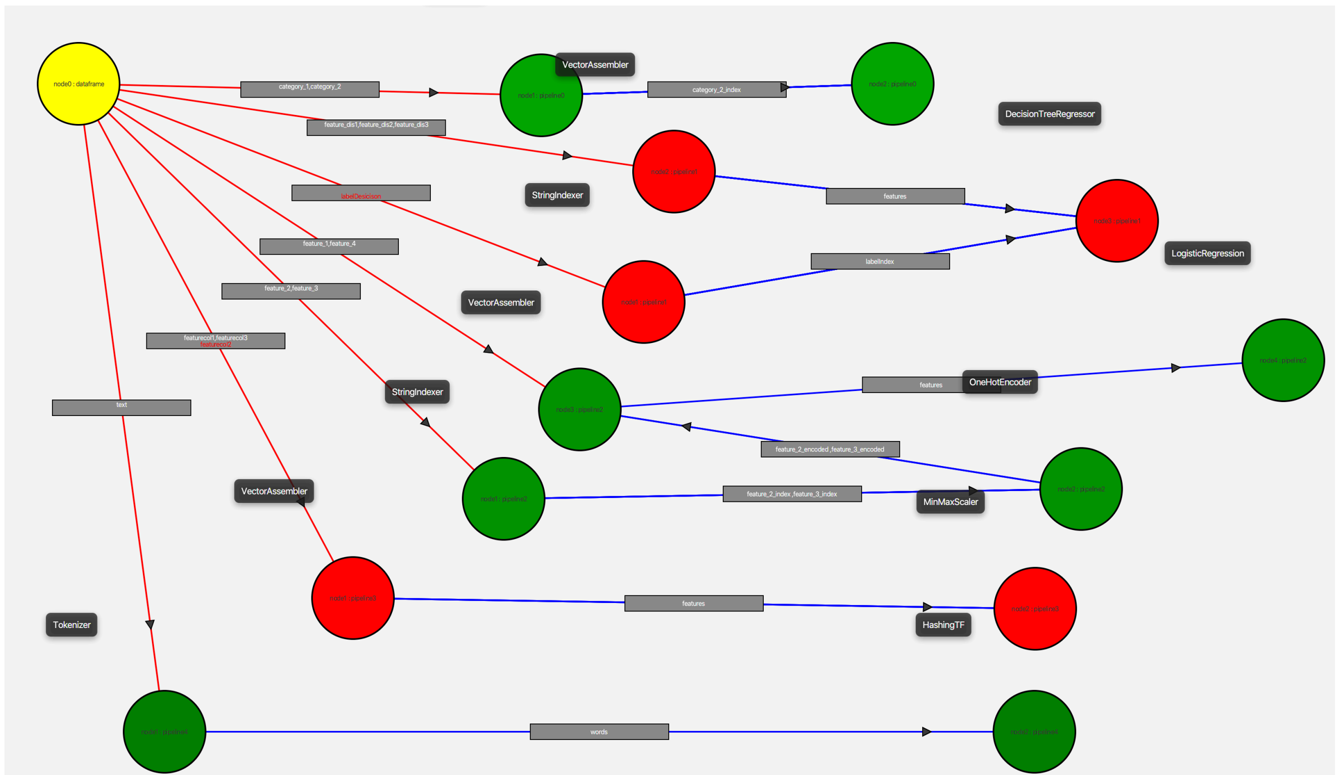

Figure 4.19: Graph after schema change

The benefits of this approach are that the analyst can be informed about the impacts of the dataframe's schema evolution to all the machine learning pipelines. Importantly, the system identifies automatically, in which parts of the pipeline is the problem and which points need to be changed so to fix its.

Chapter 5

Experiments

5.1 Time Metrics during AST Parser Execution

5.2 Time Metrics for the GraphAnalyzer Execution

In this Chapter, we report on our experimental assessment of the proposed method. We assess the two different aspects of the problem, namely the abstract representation of the pipeline as extracted from the source code and the evaluation of the impact of schema evolution.

In what follows, we will report on the time performance of the two algorithms. All our experiments are performed on a laptop with 5-core CPU running at 2.4 GHz frequency. Also memory of laptop is 8GB capacity running at 2133 MHz frequency.

5.1 Time Metrics during AST Parser Execution

First, we place our attention to the time behavior of the AST Parser. To this end, we placed timers at critical points of the AST Parser execution. Specifically, we have set the following critical points in the execution of a pipeline parsing: (a) with the term *start point* we refer to the time point when the project is been chosen for analysis; (b) with the term *end point* of the analysis, we refer to the moment when the file exporter has finished with the creation of exported file, and (c) with the term *duration* we refer to the difference between the end point and the start point. We are getting the current

time via currentTimemillis method so all values of time in our measurements have the ms (millisecond) as the time unit.

$duration = endpoint - startpoint$

Duration differs in every execution. This fact is not related with source code or the calculation way but with the scheduler of operating system. So to be more accurate at the final result, we execute every scenario of our measurement 10 times and get the average value of these 10 different values. So in this way we are more precise to our measurement.

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{1}{n}(x_1 + \cdots + x_n)$$

Above is given the mathematical formula to calculate the average value given a sum of different values. In our case n = 10.

The forces that affect the performance of the AST Parser can be listed as follows:

1. Number of Source Code Lines: the higher the volume of source code to be parsed, the longer the execution is expected
2. Number of Source Files contained in a Project: similarly, the number of source files is a homologous measure to the number of source code lines
3. Number of Pipelines: the more pipelines in the source code, requires the creation of more main memory constructs, and thus higher execution time
4. Number of Stages in Pipeline: similarly to the number of pipelines, but with a higher degree of precision

In the rest of this section we are going to analyze each one of the aforementioned factors and assess how they affect the execution time of the AST Parser.

5.1.1 AST Parser Performance over the Number of Source Code Lines

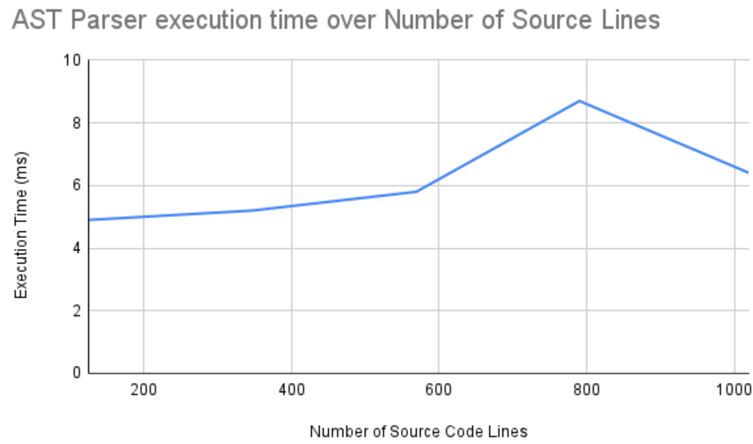

Figure 5.1: Time metrics based on Number of Source Code Lines

.

In Fig. 5.1 we assess how the number of source code lines affects the time performance of our AST parser. The horizontal axis shows a varying number of source code lines and the vertixal axis depicts the execution time for the AST parser in msec. We make our experiments for a particular set of critical values for number of lines. These critical set of values are the following: {125, 347, 569, 790 and 1020 }.

There is an almost linear correlation between number of source code lines and execution time of AST Parser when range of source code lines is [0, 1000]. With the exception of the case of 800 code lines where there is a small, unexpected increase in execution time, the rest of the experiments show a practically liner behavior of the ASTParser with respect to the processing time.

5.1.2 AST Parser Performance over the Number of Source Code Files

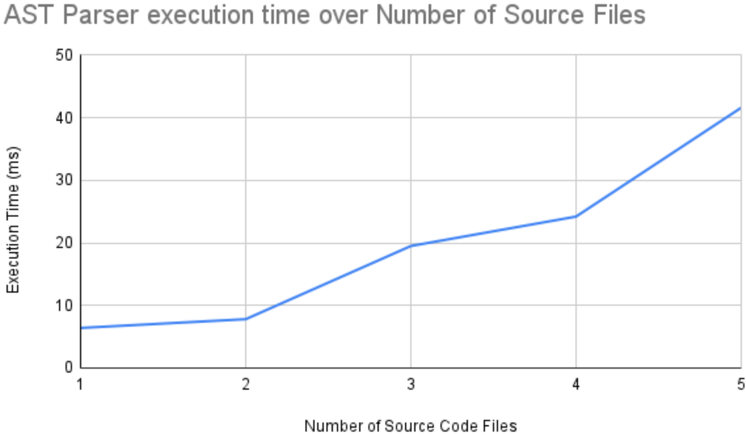

Figure 5.2: Time metrics based on Number of Source Code Files

In Fig. 5.2 we assess how the number of source code files affects the time performance of our AST parser. The horizontal axis shows a varying number of source code files and the vertical axis depicts the execution time for the AST parser in msec. We make our experiments for a particular set of critical values for number of Source Code Files. These critical set of values are the following: {1, 2, 3, 4 and 5 }.

Expectedly, there is a correlation between the number of source code files and the execution time of AST parser. Each file contains more than 1000 lines so we can conclude that number of source code lines affects the execution time when is much bigger than 1000 lines. Also it is reasonable execution time of AST parser to be increasing when source files are being populated because needs extra time for each one to check.

5.1.3 AST Parser Performance over the Number of Pipeline Stages

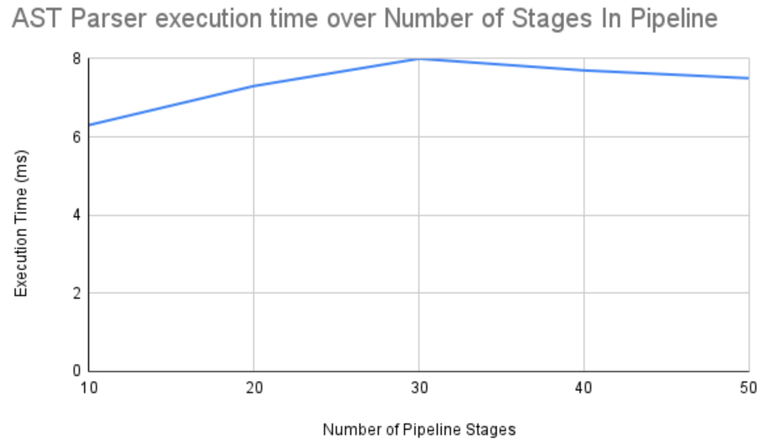

Figure 5.3: Time metrics based on Number of Pipeline Stages

In Fig 5.3 we assess how the number of pipeline stages affects the time performance of our AST parser. The horizontal axis shows a varying number of pipeline stages and the vertical axis depicts the execution time for the AST parser in msec. We make our experiments for a particular set of critical values for number of Pipeline Stages. These critical set of values are the following: {10, 20, 30, 40 and 50 }.

As we see from diagram it seems to be a linear relationship between execution time of AST Parser and number of pipeline stages until the number of pipeline stages is equals with 30. This is a critical point regarding the time as after that time is stabilized to a particular value and don't depends on the number of pipeline stages when the number of stages is in the range of [30,50].

5.1.4 AST Parser Performance over the Number of Pipelines

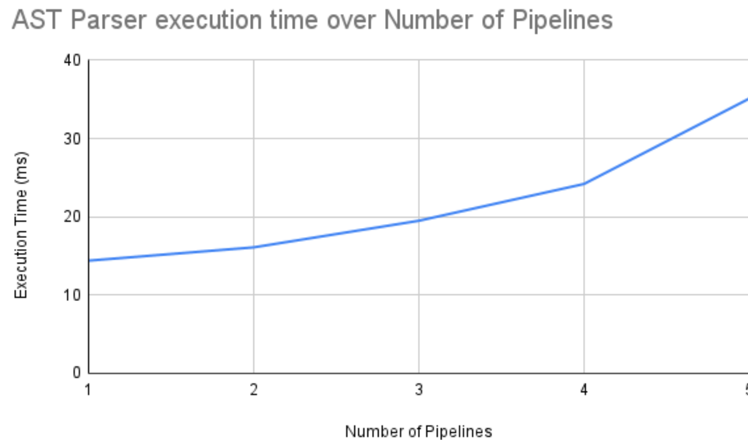

Figure 5.4: Time metrics based on Number of Pipelines

In Fig 5.4 we assess how the number of pipelines affects the time performance of our AST parser. The horizontal axis shows a varying number of pipelines and the vertical axis depicts the execution time for the AST parser in msec. We make our experiments for a particular set of critical values for number of Pipelines. These critical set of values are the following: {1, 2, 3, 4 and 5 }. As one would expect, the execution time is monotonically increasing with the number of pipelines.

5.2 Time Metrics for the GraphAnalyzer Execution

Apart from experimentally measuring the performance of the ASTParser, we are also going to examine the time behaviour of GraphAnalyzer.

We placed timers at critical points of GraphAnalyzer. Specifically, we have set the following critical points in the creation of graph: (a) with the term start point we refer to the time point when file importer starts to read the file contains all info required in the creation of graph; (b) with the term end point of the designing , we refer to the moment that analyzer has added the final graph to the JPanel, and (c) with the term duration we refer to the difference between the end point and the start point.

$$duration = endpoint - startpoint$$

Duration differs in every execution. This fact is not related with source code or the

calculation way but with the scheduler of operating system. So, to be more accurate at the final result, we execute every scenario of our measurement 10 times and get the average value of these 10 different values.

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{1}{n}\left(x_1 + \cdots + x_n\right)$$

Above is given the mathematical formula to calculate the average value given a sum of different values. In our case n = 10.

In the list below we mention a set of variables that may affect the efficiency of GraphAnalyzer in terms of execution time. The assessment of the effect of each one of them presents an experimental scenario of our measurements.

1. Number of Pipelines:
2. Number of Stages in Pipeline:
3. Type of Graph: as much complex the graph as higher execution time.

We expect the time execution of GraphAnalyzer to be significantly higher than time execution of AST Parser because GragpAnalyzer employs a variety of processes with much more demanding tasks, like reading files from disk, designing graphical representations via javafx, and applying topological sorting to the resulting graph.

5.2.1 Time performance of GraphAnalyzer over the number of Pipelines

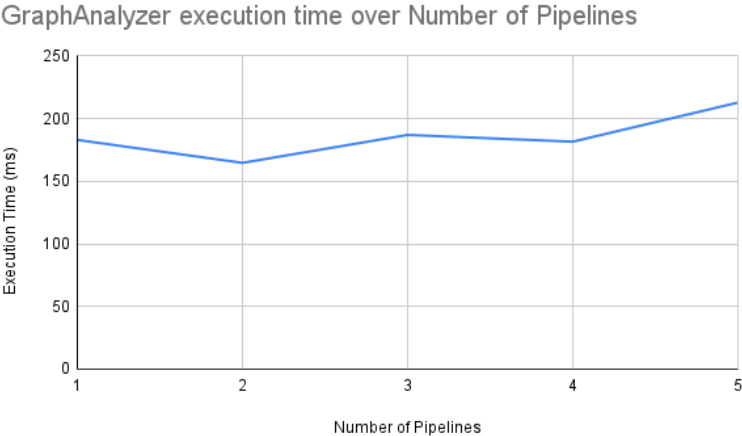

Figure 5.5: Time metrics based on Number of Pipelines

In Fig 5.5 we assess how the number of pipelines affects the time performance of our GraphAnalyzer. The horizontal axis shows a varying number of pipelines and the vertical axis depicts the execution time for the GraphAnalyzer in msec.

It seems that there is not any correlation between number of pipelines and execution time of GraphAnalyzer. This is an expected result because number of pipelines doesn't significantly affect the design of graph that takes most of the execution time.

5.2.2 Time performance of GraphAnalyzer over the number of Pipeline Stages

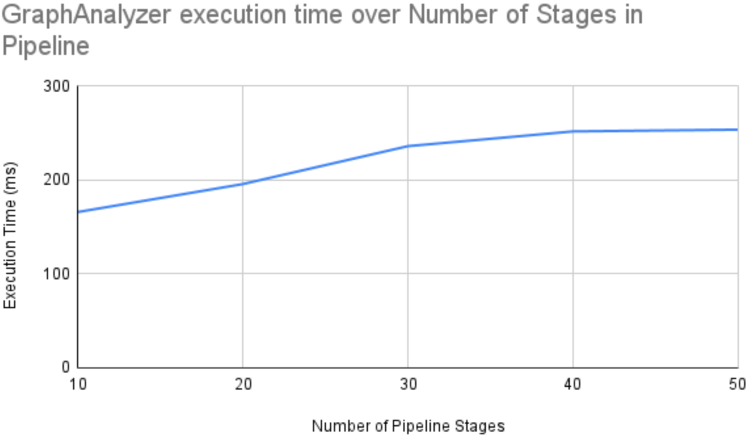

Figure 5.6: Time metrics based on Number of Pipeline Stages.

In Fig 5.6 we assess how the number of pipeline stages affects the time performance of GraphAnalyzer. The horizontal axis shows a varying number of pipeline stages and the vertical axis depicts the execution time for the GraphAnalyzer in msec.

In the diagram we see something that was expected from our side to be a correlation between number of pipeline stages and execution time of GraphAnalyzer. The more stages exist, the more lines to read from the file importer of GraphAnalyzer, and the more nodes for GraphAnalyzer to create. As the number of stages increases, there is much more time during topological sorting because algorithm sorts nodes of graph.

5.2.3 Time performance of GraphAnalyzer over the Type of Graph

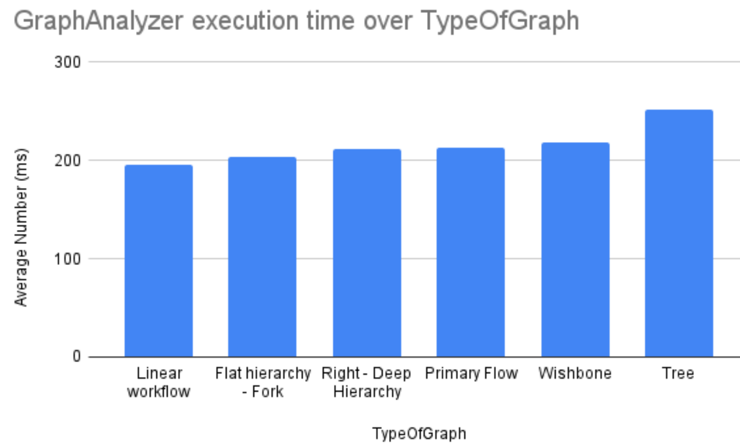

Figure 5.7: Time metrics based on Type of Graph

In Fig 5.7 we assess how the type of graph affects the time performance of our GraphAnalyzer. The horizontal axis shows a varying types of Graph and the vertical axis depicts the execution time for the GraphAnalyzer in msec.

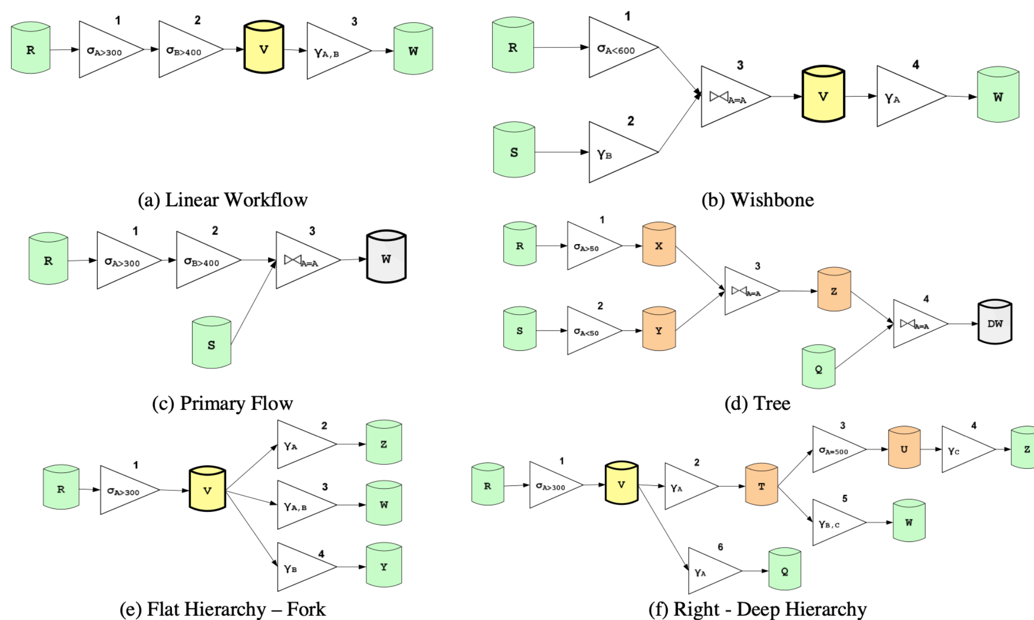

Figure 5.8: Types of Graph [13]

It is obvious that there is a correlation between type of graph and execution time of GraphAnalyzer. The increase in time is seems to be very small in all types of

graph except of Tree type. Linear workflow graph requires the minimum time while Tree graph requires the maximum time of all cases. So as much complex the graph as much time to execute it by GraphAnalyzer. Based on previous scenarios we can conclude that the worst combination regarding the time should be a graph with a lot of nodes and complicated design type.

Chapter 6

Conclusions and Future Work

The current thesis has dealt with machine learning pipelines from the viewpoint of software engineering, and provides two contributions. First, we have shown a method to provide an abstraction of the source code of the pipelines as main-memory structures. The method works by exploiting the Abstract Syntax Tree of the source code and extracting the necessary statements that define stages and pipelines, out of the entire set of statements that constitute the source code. The outcome of this process is a main-memory representation of the pipeline as a graph, with stages as its nodes and input-output dependencies between subsequent stages as its edges. The tool that we have constructed to facilitate this analysis, also allows the graphical representation of all machine learning pipelines found in the source code of a certain project.

Moreover, by exploiting this graph representation, this Thesis provides a method, via which we can assess the impact of schema evolution in a set of pipelines that all stem from the same data provider (e.g., a data file with structured records). Assuming that the structure of the records of a file is changed, with certain attributes being inserted and other attributes being deleted with respect to its previous version, the proposed method exploits the dependencies that exist between stages of pipeline and the source data, and propagates the impact of the change over the graph. As before, our tool can also visualize the impact of the change in the graphical representation of the pipeline.

There also actions that can improve much better the fact to handle changes in schema of dataframe. Firstly stages used inside machine learning pipeline are type

sensitive. That means that arguments used as inputs in pipeline stages must have a particular type. So if there is a change only in type of column the current implementation doesn't cover this case.

Moreover one suggestion should be to develop some kind of controller that will check for every argument is used in pipeline, what kind of type is expected from the stage that is used. If there is a difference between type from dataframe and expected type, controller should change it to the appropriate type so as the pipeline to not fail. Secondly in case where a column in dataframe has been changed, our application as we said before can handle this change and inform us about it and in which stage of pipeline is used. Thus, another improvement is to take advantage of AST Parser that can make changes to the existed source code and so as to change the argument of specific stage based on the change coming from dataframe.

Bibliography

[1] J. Andany, M. Léonard, and C. Palisser, "Management of schema evolution in databases," in *Proceedings of the 17th International Conference on Very Large Data Bases*, ser. VLDB '91. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, p. 161–170.

[2] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. USA: Cambridge University Press, 2014.

[3] A. Paligiannis, "A mapping study in the area of schema evolution," Univ. Ioannina, Tech. Rep., 2021.

[4] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, vol. 17, 06 2008.

[5] S. Liu, X. Wang, M. Liu, and J. Zhu, "Towards better analysis of machine learning models: A visual analytics perspective," *Visual Informatics*, vol. 1, no. 1, pp. 48–56, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2468502X17300086

[6] S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Z. Huang, "Big data analytics on apache spark," *International Journal of Data Science and Analytics*, vol. 1, no. 3, pp. 145–164, Nov 2016. [Online]. Available: https://doi.org/10.1007/s41060-016-0027-9

[7] E. Shaikh, I. Mohiuddin, Y. Alufaisan, and I. Nahvi, "Apache spark: A big data processing engine," in *2019 2nd IEEE Middle East and North Africa COMMunications Conference (MENACOMM)*, 2019, pp. 1–6.

[8] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia,

and A. Talwalkar, “Mllib: Machine learning in apache spark,” *J. Mach. Learn. Res.*, vol. 17, no. 1, p. 1235–1241, jan 2016.

[9] S. Slinger, “Code smell detection in eclipse,” Delft Univ., Tech. Rep., 01 2005. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.4897&rep=rep1&type=pdf

[10] K. D. Cooper, K. Kennedy, and L. Torczon, “Compilers,” in *Encyclopedia of Physical Science and Technology (Third Edition)*, third edition ed., R. A. Meyers, Ed. New York: Academic Press, 2003, pp. 433–442. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B0122274105001265

[11] R. M. Fuhrer, M. Keller, and A. Kiezun, “Advanced refactoring in the eclipse JDT: past, present, and future,” in *1st Workshop on Refactoring Tools, WRT 2007, in conjunction with 21st European Conference on Object-Oriented Programming, July 30 - August 03, 2007, Berlin, Germany, Proceedings*, 2007, pp. 30–31. [Online]. Available: http://netfiles.uiuc.edu/dig/RefactoringWorkshop/

[12] J. McAffer, J.-M. Lemieux, and C. Aniszczyk, *Eclipse Rich Client Platform*, 2nd ed. Addison-Wesley Professional, 2010.

[13] P. Vassiliadis, A. Karagiannis, V. Tziovara, and A. Simitsis, “Towards a benchmark for ETL workflows,” in *Proceedings of the Fifth International Workshop on Quality in Databases, QDB 2007, at the VLDB 2007 conference, Vienna, Austria, September 23, 2007*, 2007, pp. 49–60.

Short Biography

Athanasios Paligiannis was born in Kalampaka, Greece in 1992. In 2010, he enrolled in the undergraduate program of Computer Science & Engineering of the University of Thessaly (Volos) and earned his Diploma in 2016. After fulfilling the obligatory military service as a private in Research and Informatics in 2017, he started working as Software Engineer in NIKI LTD until June 2021. Moreover, in 2019 he enrolled in the post-graduate program of Computer Science & Engineering of the University of Ioannina. Since July 2021, he is working as Software Developer in P&I AG in Ioannina. His research interests are primarily focused in the area of software engineering.