AN EMPIRICAL EXPLORATION OF PYTHON MACHINE LEARNING API USAGE

by

Aleksei Vilkomir

December, 2020

Director of Thesis: Mark Hills, PhD

Major Department: Computer Science

Machine learning is becoming an increasingly important part of many domains, both inside and outside of computer science. With this has come an increase in developers learning to write machine learning applications in languages like Python, using application programming interfaces (APIs) such as pandas and scikit-learn. However, given the complexity of these APIs, they can be challenging to learn, especially for new programmers. To create better tools for assisting developers with machine learning APIs, we need to understand how these APIs are currently used. In this thesis, we present a study of machine learning API usage in Python code in a corpus of machine learning projects hosted on Kaggle, a machine learning education and competition community site. We analyzed the most frequently used machine learning related libraries and the sub-modules of those libraries. Next, we studied the usage of different calls used by the developers to solve machine learning tasks. We also found information about which libraries are used in combination and discovered a number of cases where the libraries were imported but never used. We end by discussing potential next steps for further research and developments based on our work results.

AN EMPIRICAL EXPLORATION OF PYTHON MACHINE LEARNING API USAGE

A Thesis

Presented to The Faculty of the Department of Computer Science

East Carolina University

In Partial Fulfillment of the Requirements for the Degree

Master of Science in Software Engineering

by

Aleksei Vilkomir

December, 2020

AN EMPIRICAL EXPLORATION OF PYTHON MACHINE LEARNING API USAGE

by

Aleksei Vilkomir

APPROVED BY:

DIRECTOR OF THESIS: _____

Mark Hills, PhD

COMMITTEE MEMBER: _____

Nasseh Tabrizi, PhD

COMMITTEE MEMBER: _____

Rui Wu, PhD

CHAIR OF THE DEPARTMENT OF COMPUTER SCIENCE: _____

Venkat Gudivada, PhD

DEAN OF THE GRADUATE SCHOOL: _____

Paul J. Gemperline, PhD

DEDICATION

This work is dedicated to the memory of my father, Dr. Sergiy Vilkomir.

ACKNOWLEDGEMENTS

Sincere thanks to Dr. Mark Hills. I was very fortunate to have you as my supervisor. Your supervision, patience, readiness to listen, and constant encouragement meant a great deal to me.

I thank my wife Ekaterina, and my daughter Sonia, who believed in me and supported me. Without their support, this journey would never had started.

My mother, Dr. Tetyana Vilkomir, who has endless patience and is always ready to listen and support.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Background

The question of how to improve students' preparation has been discussed for decades. There are a lot of techniques on how to help students to enhance their learning skills and practices to encourage active learning. However, the situation in Application Programming Interfaces (APIs) learning has additional problems due to the variety of APIs. Even more, there are not many resources available to learn APIs. There are a lot of APIs that are used in order to improve the development process without huge effort. Nevertheless, it takes a lot of time and effort to study how APIs could be used.

An exploratory survey [33] indicates that APIs learning resources are the main issue in the API learning experience. Reading the provided documentation could be a critical element even for experienced developers but could be a stumbling block for the students. One of the well-known approaches to simplify the usage of APIs is code completion. This approach is implemented in different Integrated Development Environments (IDEs) and often is the reason why the IDEs are used. Bruch, in his study [6], compares code completion functionality with the specific browser that allows developers to see possible APIs methods even without knowing the names of the methods. He even named code completion as one of the milestones in the

transition from old to modern IDEs.

In general, we could distinguish two types of code completion approaches that are used by most IDEs. In the first, the alphabetic list approach, the IDE provides an alphabetic list of all possible completions. In the second, "smart" auto-completion, a context-based completion is provided.

The first type is no longer commonly used; it is not helpful because useful recommendations may be quite far down the list, so order does not say anything about usefulness. The second type is successfully used in different IDEs for different languages. As a general-purpose language, Python is now one of the fastest-growing programming languages [38]. Despite the simplicity, Python provides a lot of options for the users. Because of its dynamic nature, it is not possible to apply the same code completion approaches as are used for static languages. That is why there is room for applying a new methodology.

Machine learning approaches, such as maximum entropy, genetic algorithms, random forest, and neural network, have been used in different application areas with great success. Machine learning is one of the areas where Python proved itself as a very simple but powerful tool. According to GitHub, based on GitHub activity in 2018, Python is the most commonly-used language for machine learning projects [12]. Popular Python libraries for machine learning and data science include matplotlib [22], numpy [28], pandas [29], scikit-learn [36], and scipy [37]. That is why it is widely used not only by software developers but by scientists and students. That is why it is important to provide an "intelligent" completion method that would provide auto-completion based on the context of the code, not a traditional sequential list of functions. It is important because some users are less sophisticated or bring less programming knowledge.

1.2 Motivations for this study

I have studied several programming languages and techniques during the two last decades. That is why I clearly understand that it is very hard to start developing software without any help. Even more, existing code completion tools are not effective for this work's research area due to the following facts:

1. General API recommendation tools would provide a general recommendation, not machine learning-oriented.

2. Alphabetic recommendation lists are too long and would not help if a user is not skilled enough.

3. Existing "intelligent" systems are mainly using code source from different application areas, not specific for machine learning.

1.3 The purpose of this study

This empirical study aims to understand how Machine Learning APIs are used by data scientists to solve various machine learning tasks. The study also sought information that could provide additional support for the future development of a recommender system dedicated to helping students and other novice developers to learn the usage of ML APIs.

1.4 Research Questions

In order to get an understanding of the APIs typically used to solve machine learning tasks, we defined the following questions to be answered:

- **RQ1:** What are the main programming languages used for data science applications?

- **RQ2:** What are the main machine learning libraries used in Python machine learning code?
- **RQ3:** In each of these libraries, what are the most used modules?
- **RQ4:** For each of the libraries and modules, what are the most common API calls?

Answering these questions gives us insight into how machine learning APIs are used in practice. It is important to know what are the main languages applied to data science problems in order to know where to focus further research and for which language recommender system could be useful. This also helps to ensure the focus of this thesis, which is on API usage in Python, actually targets a common platform.

The APIs libraries could have different structures and could have different modules, and knowing those modules could help us to find the patterns of their usage. Finally, empirical data about calls to machine learning APIs, including call frequency, could serve as the basis for future recommender system.

1.5 The structure of the thesis

The rest of the thesis is structured as follows. First, related work is described in Chapter 2. Next, we describe the current works in the area of APIs and code completion systems. Then, in Chapter 3, we introduce a corpus of Python code and describe the source of the data and ways of its extraction. Further, in Chapter 4, we describe the methods and processes used for extracting data from the corpus. Chapter 5 presents the results of our work together with the answers to the research questions. Finally, in Chapter 6, we conclude our work and propose future work.

1.6 Contribution

There have been a number of studies of how APIs are used by developers, which we discuss in Chapter 2. However, to the best of our knowledge, none of these studies focus on the usage of machine learning APIs for Python. We plan to use the results of this empirical study as a foundation for the future development of an API recommender system that would help students and other novice developers learn how to properly use machine learning APIs in Python.

Chapter 2

Related Work

In this chapter, we first present the information about the existing work related to APIs usage in Section 2.1. Next, we discuss existing approaches and works in the field of code completion systems in Section 2.2.

2.1 APIs usage and learning approaches

The Application Programming Interface term was introduced in 1975 by Date [8]. However, active modern APIs usage started in the early 2000s. emThe largest APIs directory [1] had reached the size of more than 23,000 APIs in September 2020. Almost every day, a new API is introduced for public usage [2]. Sylos and Myers [24] states that almost every line of the code written by the developer could contain a reference to one of the APIs. Due to continuous changes in the APIs' tasks and improvement, it becomes very complicated to use APIs effectively. That is why APIs usage becomes not a useful interface between them and the code but an additional learning task.

Robillard [34] conducted different series of studies, including surveys and interviews, to find out existing issues faced by the programmers while learning and applying APIs. This study found out that the developers are not only challenged with finding some patterns or scenarios of APIs usage but have some complications with

the understanding of background processes of APIs usage. Due to these issues, developers could have problems choosing the correct methods and classes from APIs. Even more, they could have an additional problem when trying to combine different methods and classes from different parts of APIs. Despite the fact that APIs' names could be self-explaining, it might be too complicated to choose the correct ones.

There were several studies done in the area of improving APIs learnability[11, 39, 40]. During this studies some of the obstacles were outlined: awareness about the structure of the APIs, picking out the classes and methods that are suitable for the tasks that are solved, finding the correct approach to use the classes, and combining the usage of different objects, especially from different APIs. One of the approaches to improve the usage of APIs was a design improvement. Stylos [24] proposed to improve the design of the APIs in order to include usability as one of the significant optimization parameters for all APIs, and they anticipated that this approach would be used by most of the developers. However, it is hard to believe that all developers would follow this strategy during API creation; at least it is still not the case.

Xia et al. [49] conducted a survey in which they asked 235 software engineers to rate a set of search tasks. They noted that almost 10% of all analyzed search queries were related to reusable code snippets. Participated software engineers stated that they do such searches as they do not always remember the API usage's correct structure. They do not want to redo some tasks that were already implemented by somebody else. However, the developers also highlighted that it is not easy to find the exact snippet. The search engines often skip special symbols that could be an essential part of the snippet. That is why the results of the search could lead to the wrong examples. They also complained about the snippets' quality and correctness that could be found using a simple Web search. In the earlier study [4], that tracking system was used to evaluate the activities of the developers. The results demonstrated

that a software engineer could make more than 20 API related searches per day.

There were several pieces of research made on the APIs usage based on the Java APIs. Zhong and Mei [51] presented a study about APIs usage based on the seven projects that were extracted from SourceForge[1] and Apache[2]. All those projects were written using Java. The authors noticed that the different types of API libraries can have different usage and needed to be studied separately. Nguyen et al. [25] introduce the idea that it is possible to distill preconditions of the APIs usage from a large data corpus. They tested this idea by mining preconditions for Java Development Kit from more than 100 million SLOC. Sawant and Bacchelli [35] developed fine-GRAPE, a method that extracts exact API usage information taking into account type information. However, those researches do not answer the question of how developers use the APIs from a machine learning point of view. Even more, not all approaches used for static languages could be applied to the dynamic ones.

Hora [16] presented various limitations of existing websites that provide examples of APIs usage. First of all, the author refers to the examples' quality — often, it is hard to get a full understanding of the example and reuse it. Next, those websites contain duplicated and irrelevant examples. This fact makes it harder to find the information that is needed to solve a problem. Finally, some examples are created manually, and it makes it impossible to cover a wide range of existing APIs. To overcome those limitations, the author presents the APISonar approach, which extracts API usage examples from a set of projects and creates a ranked representation of those examples.

[1] http://sourcefourge.net/
[2] http://www.apache.org/

2.2 Code completion systems

From the beginning, code completion was presented as an alphabetically sorted list of all possible methods. Robbes in [32] presented an approach to improve code completion with program history. The authors demonstrated that usage of different types of program changes could be successfully used to improve code recommendations. Bruch et al. [7] introduces best matching neighbors algorithm to improve the k-nearest neighbor algorithm for discovering applicable recommendations for target objects. Hindle [15] used the n-gram model to prove that code could be modeled by statistical language models and that those models could be used in order to support Java software developers. The authors presented their auto-completion plug-in for the Eclipse. Tu and Su [48] introduced a cache language model that extends the n-gram model with the cache to exploit localness – local patterns in the code. Hellendoorn [14] suggests that a scope-based model with an unlimited vocabulary could significantly outperform existing k-gram models, as well as RNN and LSTM deep-learning language models. Nguyen and Nguyen in [26] introduce the statistical semantic language model (SLAMC). This model proposes relying on the lexical analysis to capture the patterns in the code and take into account the semantic, which is well defined in the programming languages. D'Souza et al. [10] proposed an approach that extends the BMN algorithm with additional code usage frequency. They assume that the BMN algorithm outperforms methods used by association-rule mining. This work was applied to the Python language and showed promising results recommending code. The proposed methodology is used in our work as a starting point.

There is another group of approaches that are used for code recommendation. This group is based on neural networks and deep learning. Bhoopchand [5] introduced a model that analyzes the introduction of identifiers based on the AST examination.

This model is a pointer network that allows us to study long-range dependencies in the Python code. Li and Wang [21] present an approach based on the same model but target a prediction of out-of-vocabulary words. The pointer network in this approach could either generate the word in vocabulary using the Recurrent Neural Network component or regenerate an out-of-vocabulary word using the pointer component. Karampatsis [20] presents a corpus of 13,000 projects that are used for studying different modeling approaches. The authors introduce an open vocabulary source code natural language model and apply it to the Java, C, and Python code. Svyatkovskiy et al. [42] presented Artificial Intelligent assisted code completion system for Python code. This system is designed to be a part of Microsoft Visual Studio IDE. Pythia uses Long Short Term Memory networks. The networks are trained on snippets that are extracted from the open-source code dataset. In his next [43] work, he uses a similar approach, introducing a ranking mechanism. The authors patented [41] their system in 2020. Asaduzzaman et al. [3] presented a CSCC (context-sensitive code completion) that is an example-based completion tool. The system was used for Java language and is not in the scope of our work.

Finally, several research pieces were presented based on the mining of API usage from open source Java projects. Zhong [52] developed Mining API usage Pattern from Open source repositories (MAPO) tool. This tool mines frequent sub-sequences from the code snippets and make automated recommendations. The tool was integrated as a recommender for Eclipse IDE. Xu et al. [50] introduced their Method usage and Location for API (MULAPI) approach that uses feature location together with the historical feature repository to provide features related recommendations. Niu et al. [27] described the approach of mining API usage patterns for the recommendations. They targeted the mobile application development domain in their work. The set of more than 11,000 Android programs was used to make an empirical study to confirm

that their approach can effectively mine API patterns.

Chapter 3

Corpus

To investigate the use of machine learning APIs in Python code, we first needed to identify a corpus of Python code focused on machine learning that could be analyzed. Many similar studies use repositories hosted on GitHub. For this study, we decided this would not be feasible: we would first need to identify repositories that use Python, then determine if the code in each repository was focused on machine learning. It could be possible to do this, e.g., by filtering repositories by the libraries used, but this also risks mixing code for machine learning with code focused on other program concerns.

Because of this, we opted to use Kaggle [18]. Kaggle is a community site for machine learning researchers and practitioners, providing courses, datasets, and discussion boards. Kaggle also hosts machine learning competitions, where different Kaggle members can submit their solutions to solving posted problems. Kaggle supports multiple languages, including Python. Details on existing competitions, and publicly-posted solutions, can be found on the Kaggle website and in the Meta Kaggle dataset [23]. Here, we use Meta Kaggle to identify posted solutions written in Python, which are then downloaded using the Kaggle API.

3.1 Kernels extraction

The Meta Kaggle set does not have the source code in it. The data set contains 29 CSV files; the total size is 8.66 GB. There are seven main entities, each of them presented by their own file: Competitions, Kernels, Forums, Tags, Teams, Users, and Kernels. All other CSVs presents relationship tables between the entities. For our research, the only important tables are the following: Kernels, Users, KernelLanguages, and KernelVersions. Kernels table contains attributes of the kernels - source code files. Users table, obviously, includes information on users - authors of the source code files. KernelLanguages contains the list of programming languages that are used in the kernels. Finally, KernelVersions provides us with the possibility to find out which kernel uses which programming language. Even more, this table allows us to have an understanding of used programming languages without having to download the code files first.

In order to get the public source code, it was necessary to use Kaggle API. Kaggle API is accessible using a command-line tool written using Python 3 and provided by Kaggle. This means that the only way to download source code files is to do it one by one from the command prompt. That is why we decided to create a script (.cmd) file with a command for each file we want to download.

In order to create such a command file, the Python script (Figure 3.1) extracts all kernels that are written using the selected languages. The values 2,8 in line 7 presents Python and Jupyter Notebooks correspondingly. The kernels are sorted by the total amount of votes each of those received (line 8). Finally, the results are truncated after 100,000 kernels in order to get a representative sample but have a reasonable processing time.

The user names for the selected kernels are extracted from the Users table using

the authors' IDs provided in the Kernels table (line 15). Those names are combined with the URLs of the corresponding kernel files and add to the script file (line 17).

```
1  import pandas as pd
2
3  kernels = pd.read_csv("Kernels.csv")
4  kernelsversions = pd.read_csv("KernelVersions.csv")
5  selectedkernels = kernels[kernels.Id.isin(
6      kernelsversions[kernelsversions.
7                      ScriptLanguageId.isin({2, 8})].ScriptId)].\
8                      sort_values('TotalVotes', ascending=False)\
9      .head(100000)
10
11 users = pd.read_csv("Users.csv")
12 with open("get_kernels.cmd", "w+", encoding="utf-8") as file:
13     for i,j in selectedkernels.iterrows():
14         commandstrings="kaggle kernels pull " +\
15         users[users.Id==j.AuthorUserId].UserName +\
16         "/" +j.CurrentUrlSlug +"\n"
17         file.writelines(commandstrings)
18     file.close()
```

Figure 3.1: Batch generation

This script applies Kaggle API methods ("kaggle kernels pull") for downloading of the source code files. This approach is highly time-consuming. It required 3.5 days to execute all script commands and download selected sources. However, this is the only available way to download Kaggle source files.

In some cases, projects that were in the top 100000 were not downloaded. This happened in the following cases:

- Some of the projects were set up as private. This means that during our download process, those files were skipped.
- Some of the files that were referenced by Meta Kaggle were removed.
- Some of the projects have more than one vote despite the fact that they were empty. We have to drop those files as well.
- Some projects were processed by Kaggle API with an error, and those files were not downloaded.

Our approach requires source files to be pure Python files. However, 93% of the projects are done using the Jupyter Notebooks [17]. From the file structure point of view, Jupyter Notebooks are JSON documents, which contain not only source code, but also formatting, metadata, and media output produced by the code execution. This means that we had to convert IPython Notebooks to the plain Python code.

We decided to use the standard Jupyter mechanism to convert the files - *nbconvert*. While applying this mechanism, another unexpected issue was found. The Kaggle competition was done by data scientists from all around the world. Some of the scientists decided to use their native languages in the source code. The Jupyter method was not able to convert the files that contained special characters (e.g., ä, ü). That is why we had to apply Microsoft PowerShell 7.0 script to convert all notebooks files to the UTF-8 encoding standard. This script iterates through files in the folder, extracts the content of each file, and overwrites the content with the same content converted to UTF8 encoding.

However, this convertion did not help to eliminate all possible errors produced by nbconvert. That is why this mechanism was not able to process all files and crashed before completing the task. We had to use the same brute-force approach we used before and create a command file with the nbconvert call for each notebook file.

As a result of all described above, the collection of 69,376 source files (9,487,108 lines of code, ignoring more than five million lines of comments and more than eight millions blank lines) was used for further processing. The average size of the file is 138 LOC. The maximum file size is 808 LOC; the minimum size is 5 LOC.

3.2 Competitions extraction

The Meta-Kaggle does not contain a complete database of all competitions that took place. The description of this data set specifically states that this set is not a complete dump of the Kaggle database. The data in the tables is filtered out, and some rows and columns are transformed.

That is why it was not possible to extract all connections between source-code and competitions. In order to do this, we have to create an approach (Figure 3.2) to extract as many competitions as possible. Most of the source-code files have a web-page that could be open using the name of the kernel. Most of the web-pages contain information from which competition the data was used. However, there is no straight-forward way to get this information from the page.

Following our approach, the URL of the target page was generated (Figure 3.2, line 6). The web page is open, and its content is extracted (line 8). We used a Python HTML parser to parse the content to the XML (line 9). The target name of the competition is extracted from the XML using the HTML tag name (lines 10 - 12). In case there is no information (line 15) or encoding error (line 17) - "No competition" is used as a competition name. Finally, a pair (source_code name, competition) is stored in the CSV file.

As a result, we extracted 8,485 different names of the competitions. However, four of the top ten (3.1) used competitions are "multiple data source", "no data source", and "no competition". That is why the number of competitions presented above is the minimum quantity. More than 50% of the competitions (4,907) are presented by only one project.

From the project count point of view, top 10 competitions are connected with more than 40% of the projects (41,167). Almost 40% of the projects are using either

multiple or unavailable for us to reveal competitions.

```
with open("kernels_competitions.csv", "w+", encoding="utf-8") as
    file:
     for i, j in selectedkernels.iterrows():
         competition_count += 1
         uid = users[users.Id == j.AuthorUserId]
         if uid.size > 0:
             url = 'https://www.kaggle.com/' + uid['UserName'].iloc
    [0] + '/' + j.CurrentUrlSlug
             try:
                 webpage = urlopen(url)
                 soup = BeautifulSoup(webpage, "lxml")
                 description = (soup.find("meta",\
                 property="og:description")).\
                 encode('utf-8', errors='ignore').strip()
                 competition = description.split("Using data from "\
                 .encode('utf-8').strip(), 1)[1][:-29]\
                 if description else "No competition"
                 competition = competition.decode('utf-8')
             except (URLError, UnicodeEncodeError) as e:
                 competition = "No competition"
             file.writelines(j.CurrentUrlSlug+","+competition+"\n")
             print(competition_count)
     file.close()
```

Figure 3.2: Source-code - competitions extraction

As a result of all data manipulations, we got a set of Python (*.py) files that contains a source code written by data scientists or data enthusiasts. Furthermore, we have a list of the file names and corresponding competition names. Due to the fact that the Kaggle Meta-Data is a live data set and it could be modified on a daily base, and there are no daily versions available, we decided to provide the source files (as a zip archive) and the list of kernel-competition (as CSV file) in the GitHub repository[1].

3.3 Summary

We have collected 69,376 Python script files dedicated to the data science tasks solving from the Kaggle competitions. The overall SLOC is 9,487,108. We have distilled 8,485

[1]https://github.com/ecu-plse-lab/Python-MLAPI-expl

Competition name	Projects count
multiple data sources	15975
Titanic: Machine Learning from Disaster	6058
no data sources	5466
Private Datasource	4880
Digit Recognizer	2511
House Prices: Advanced Regression Techniques	2346
No competition	1205
Housing Prices Competition for Kaggle Learn Users	1188
Iris Species	815
Credit Card Fraud Detection	723

Table 3.1: Top 10 used competitions

of different competitions names.

Chapter 4

Research Method and Data Extraction

The usage of APIs could be a challenging task for software engineers. That is why it is important to support API usage studying process for the student. However, before creating a system that would support students it is necessary to understand how the data scientists are using Machine Learning APIs.

This chapter is focused on the methods we are using to extract the data from the Python sources. The data is extracted from the corpus described in the Chapter 3 and is analyzed in the Chapter 5. In order to extract the list of modules it was decided to use Abstract Syntax Tree (AST) visitor approach. An abstract syntax tree is a finite, labeled, oriented tree in which internal nodes are associated with programming language operators, and leaves with corresponding operands. A visitor goes through nodes and if the visitor method is specified for the node - it is executed.

The list of modules used in *import* and *import from* statements are mined from each source code. The total number of usages per module is analyzed and summarized afterwards. Such a walk-through mechanism could be implemented with not very complex code (Figure 4.1). We implemented visit_Import method (lines 1-4) that would be called for each ***import*** *libraryname* and ***import*** *libraryname* ***as*** *alias* statement in the Python code found by visitor. Additionally, we implemented visit_ImportFrom (lines 6-9) method that would be executed for each statement ***from***

libraryname ***import*** *submodule*. Each method stores the name of the library (lines 3,8) in the dictionary that could be used for the further processing. In case of multiple libraries are imported (e.g., from sklearn.pipeline import Pipeline, FeatureUnion) the iteration mechanism is implemented (lines 2,7).

```
1 def visit_Import(self, node):
2     for alias in node.names:
3         self.stats.append(alias.name)
4     self.generic_visit(node)
5
6 def visit_ImportFrom(self, node):
7     for alias in node.names:
8             self.stats.append(alias.name)
9     self.generic_visit(node)
```

Figure 4.1: AST visitor imports

During our work we did not change any code in the original source file. The only alteration that was produced was change of the encoding of the Jupyter Notebooks as described in Chapter 3. All script files, that were used for the data extraction and analyses could be found in the repository of our work, under the Scripts folder.

4.1 Modules usage

The main disadvantage of the standard AST visitor method is the fact that modules could be used (imported) in different, not necessary straight forward ways. There are five ways[30] to implement import in a Python code (Figure 4.2). This fact means that we could lose a lot of modules calls with the described above approach. Taking

```
1 import foo                    #Direct import of the foo liblrary
2 import foo.bar.baz            #Sub-library foo.bar.baz is imported
3 import foo.bar.baz as fbb     #foo.bar.baz imported and bound as fbb
4 from foo.bar import baz       #foo.bar.baz imported and bound as baz
5 from foo import attr          #foo imported and foo.attr bound as attr
```

Figure 4.2: AST walk-through imports

this into the account, the new approach was used. For each import statement in the source code the root module is extracted and saved regardless the way an import was implemented (Figure 4.3). This information was used for the further modules usage

```
1    def visit_Import(self, node):
2         for alias in node.names:
3             self.stats.append(alias.name.split(".", 1)[0])
4         self.generic_visit(node)
5
6     def visit_ImportFrom(self, node):
7         for alias in node.names:
8             module_name = node.module.split(".", 1)[0]
9             self.stats.append(module_name)
10        self.generic_visit(node)
```

Figure 4.3: Updated AST visitor imports

analysis. Additionally, we decided to extract and store an information if the module was used in the project or not. This information will be used to analyse a number of useless imports.

4.2 Methods usage

For each project source code file, the list of imported libraries was used. The actual libraries' names were associated with the aliases those are bounded. When a method was analyzed, a root library was found to which this method belongs to. The method was added to a list corresponding with the root library(Figure 4.6). All libraries' lists were appended to the dictionary. Each file's dictionary was appended to the global dictionary set. The main issue was to create a mechanism to extract the root modules of the methods. The methods could be called in different ways. One of the possibilities - the method is called from the alias (Figure 4.4, line 2). Another way would be to call the method using the name of the sub-library (Figure 4.4, line 5). A third way of the usage is importing the library itself and calling methods using the

```
1 #Call from alias (example from 0-29-public-lb-score-beginner-nlp-
    tutorial.py)
2 pd.set_option('display.float_format', lambda x: '%.6f' % x)
3 ......
4 #Call from sub-library (example from 0-29-public-lb-score-beginner-
    nlp-tutorial.py)
5 Y_train = LabelEncoder().fit_transform(X_train['author'])
6 ......
7 #Call from module (example from kernel576babdf33.py)
8 battles = pandas.read_csv('/kaggle/input/final-fantasy-tactics-
    battles/fft_red-battles.csv')
```

Figure 4.4: Different ways of method calls

library's name (Figure 4.4, line 8).

For the research on the methods used we continued to use the approach based on the AST visitor (Figure 4.5). This code example demonstrates how the methods

```
1     def visit_Attribute(self, node: ast.Attribute):
2         if None != node.value and isinstance(node.value, ast.
    Attribute):
3             self.maintain_Attribute(node, node.value)
4         elif node.value is not None and isinstance(node.value, ast.
    Name):
5             if node.value.id in self.module:
6                 self.results[node.value.id].append(node.attr)
7             else:
8                 for keyAs, valueAs in self.importsAs.items():
9                     if node.value.id in valueAs:
10                        self.results[keyAs].append(node.attr)
11        elif node.value is not None and isinstance(node.value, ast.
    Call):
12            self.maintain_Attribute(node, node.value)
13
```

Figure 4.5: AST visitor code example

are checked and extracted. The visitor method for the Attribute node is defined. Visitor will enter this method for each Attribute node. First, it checks is the value of the visited node is a Name (line 4) (**pd**.read_csv()) or also an Attribute (line 2) (keras.**utils**.to_categorical(y_train, num_classes)). If it is and Attribute - the main-

```
{    'pandas': [
         'read_csv',
         'read_csv',
         'read_csv',
         'DataFrame',
         'DataFrame'
         'isna'
     ]
}
```

Figure 4.6: pandas dictionary example

tainAttribute method is called. This method is used to process Attribute nodes and to take care of the nested attribute calls. In case it is a Name - it's id that represents the name of the library, is compared with the Top 5 libraries names (line 5-10). The both cases of the import approaches are taking into the account - *import* (line 5) and *from import* (line 8). In case it is one of the Top 5 modules - the method is stored in corresponding dictionary. The maintainAttribute method iterates attribute calls up to it comes to the root name and follows the same logic we described for the Name.

Chapter 5

Data Analysis

In this chapter, we answer the questions posed in Chapter 1, based on the data extracted from the corpus described in Chapter 3. We also present a similarity analysis based on the sample of source code files dedicated to solve the same Kaggle competition. Finally, we presents the threats to validity and describe how those are mitigated.

5.1 RQ1: Programming language usage

There was a questionnaire [19] presented by Kaggle in 2018. One of the questions was "What programming language do you use on regular basis?". This questionnaire was used to confirm that the python should be used as the target for our study. This questionnaire was answered by 23,859 data science professionals. The percentage of the answers on this question are presented on Figure 5.1. The respondent were able to choose several languages in their answers.

As we can see, Python is the most used language with a wide margin comparing to the others. Even more, the analyses of the data from KernelLanguages table (described in Chapter 3) presented us that only two language families are used by the Kaggle users. Those languages are Python and R. The results of the query on the Meta Kaggle provides us with the quantities of the projects that are fulfilled

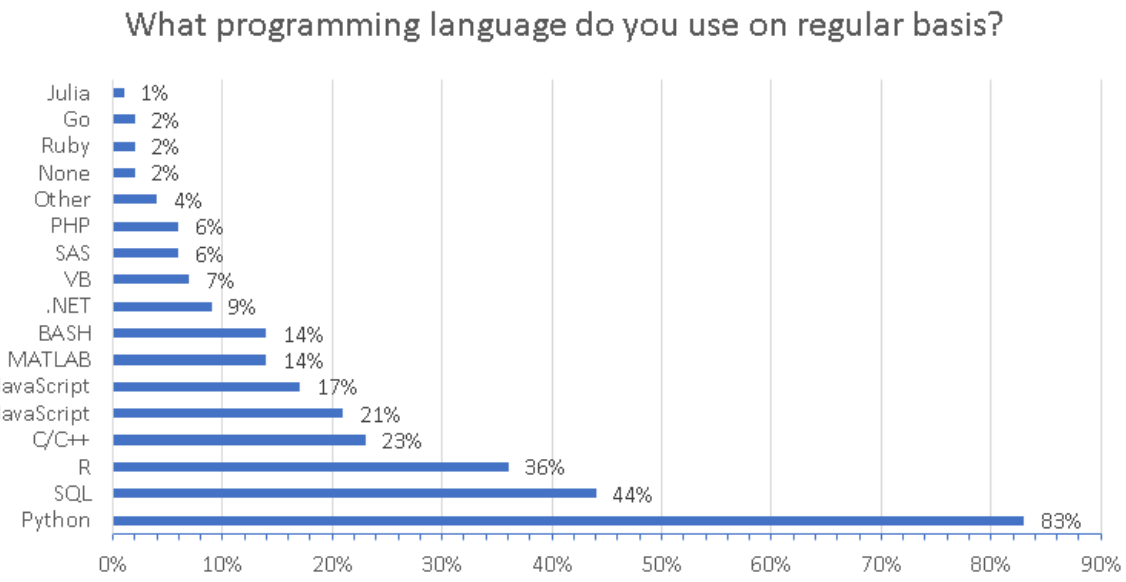

Figure 5.1: Programming languages used by data scientist, based on Kaggle survey

using those two language families (Figure 5.2). The Python languages were applied in 539,018 projects; R languages were used in 59,155 projects.

5.2 RQ2: Top 5 libraries description

NumPy allows for very efficient handling of multidimensional arrays. Many other libraries are built on NumPy, and without it, it would be impossible to use pandas, Matplotlib, or scikit-learn - which is why it ranks first on the list. It also has some well-implemented methods, such as the random function, which is much better than the standard library's random number module. NumPy is used 72,299 times in our corpus.

Data analysts typically use flat spreadsheets such as those found in SQL and Excel. Initially, this was not possible in Python. The **pandas** library allows us to work with 2D tables in Python. This high-level library allows us to build pivot tables, highlight columns, use filters by parameters, group by parameters, run functions

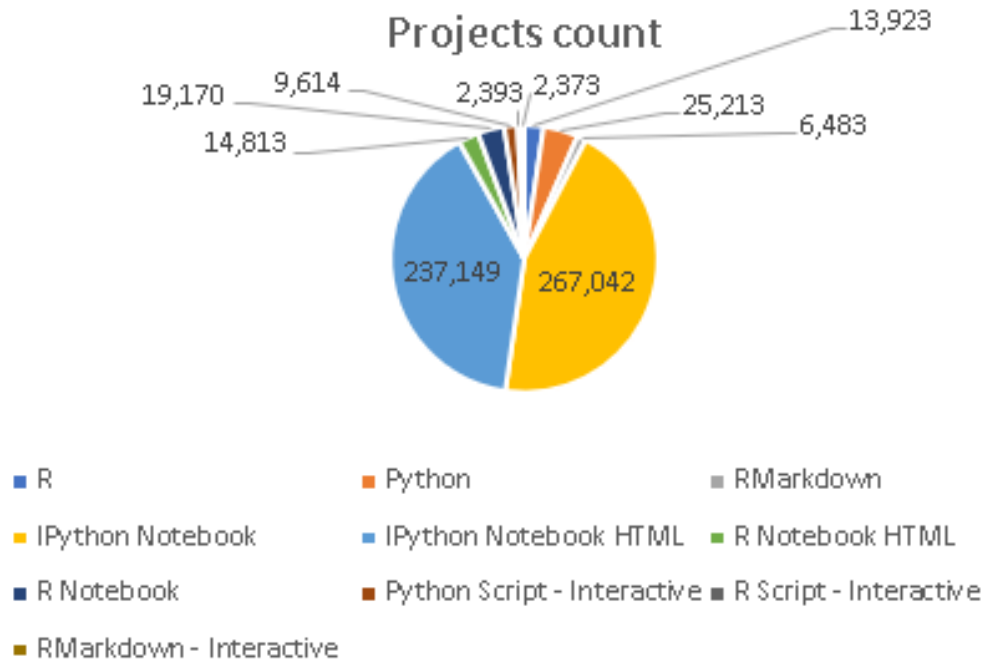

Figure 5.2: Quantity of projects per programming language in Kaggle competitions

(addition, median, average, minimum, maximum values), merge tables, and much more. Multidimensional tables can also be created in pandas. Our research shows that pandas is used 73,032 times.

Data visualization allows us to present it in a visual form, study it in more detail than can be done in a conventional format, and present it to other people. **Matplotlib** is the most popular Python library for this purpose. The best description of the library is it's motto : "Matplotlib tries to make easy things easy and hard things possible"[1]. It's not that easy to use if we want to do it in the full capacity, but with the 4-5 most common code blocks for simple line charts and scatter plots, data scientists can learn how to create them very quickly [47]. Matplotlib was used 60,775 times.

Some authors [31] consider machine learning and predictive analytic to be the most interesting features of Python, and **scikit-learn** is the most suitable library for

[1]https://matplotlib.org/3.1.0/index.html

this. It contains a range of techniques that cover everything you need for the first few years of your data analyst career: classification and regression algorithms, clustering, validation, and model selection. It can also be used to reduce the dimension of data and highlight features. Machine learning in scikit-learn is all about importing the correct modules and running a model-fitting method. It is more difficult to clean out, format, and prepare the data and to find the optimal input values and models. This module was used 102,108 times.

Finally, **Keras** is a Deep Learning API that allows developers to use backend from two more complex libraries, Theano[45] and TensorFlow[44]. It is modular based and provide simple way to switch between those two libraries or even use both together. The Keras library is most used framework that is used by the winning teams on Kaggle. In our research, Keras was used 102,108 times.

5.3 RQ3: Modules usage analysis

Based on our first approach, the initial analysis was made. We analyzed the usage of modules by direct imports. As a result, the *NumPy* module was the most used one, with a result of 60,091. The second most used module is *pandas*, with 69,706 counts. The next analysis was based on the second approach. We analyze all libraries imported in all possible ways. The top 5 used libraries and their counts are presented in Table 5.1. The complete list of the libraries used in the source files could be found in our repository.

We have analyzed all projects by the modules that are imported in those. We found out that almost 79% of projects use *pandas* module. 53% of the projects use *Matplotlib*. *NumPy* is used by 56% and *scikit-learn* is used by 46% of projects. The "worst" result was demonstrated by *Keras* - it is used by 13% of the projects. To

Module name	Module count
scikit-learn	230,633
Keras	102,108
pandas	73,032
NumPy	72,299
Matplotlib	60,775

Table 5.1: Top 5 used modules

```
# machine learning
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC, LinearSVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.cross_validation import cross_val_score
from sklearn import cross_validation
from sklearn.metrics import accuracy_score
from sklearn.cross_validation import train_test_split
from sklearn.feature_selection import SelectFromModel
from sklearn.grid_search import GridSearchCV
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import make_pipeline
from sklearn.feature_selection import SelectKBest
from sklearn.cross_validation import StratifiedKFold
from sklearn.ensemble.gradient_boosting import
    GradientBoostingClassifier
from sklearn.ensemble import ExtraTreesClassifier

```

Figure 5.3: Excessive usage of the imports (from sklearn-logisticregression-enhanced)

get a better understanding of the possible ways to combine libraries, we analyzed the usage of different combinations of the libraries.

The quantity of the projects that use all five top libraries is 2,987 (4.5%). 35,201 (52%) of the projects are using both Numpy and pandas. 23,175 (35%) projects are using Numpy, pandas, and Matplotlib libraries. 22,113 projects use Numpy, pandas, and scikit.

The most contradiction for us was the fact that one of the libraries has almost the same quantity of usages, as the quantity of the imports. However, after double-

checking the analysis approach, it was decided to check the source-code files that are analyzed. We found out that some data scientists just copies the block of codes without even considering if it would be used in the program. The example of such a block is presented in Figure 5.3. There are 17 sub-libraries imported from the *sklearn* module. Nevertheless, only four imported libraries are used in the program code.

This analysis suggested to us that some of the modules could be over-imported - imported but never used. Based on the modules, we have updated our code in order to track such cases. The AST visitor was extended to record all *from * import ** and *import * as ** imports in each file. Those are checked during walk-through and are recorded if no usage was found. The total count of the excessive imports is recorded as well as the fact if each of the Top-5 modules was imported without any reason (Table 5.2).

File name	Excessive count total	NumPy	pandas	scikit	keras	plot
0-0431-validation-rmse.py	3	1	0	1	0	0
0-1-data-science-introduction-heart-diseases.py	1	1	0	0	0	0
0-1-house-prices.py	6	0	0	1	0	1
0-11701-top-10-57-with-only-one-entry.py	0	0	0	0	0	0
0-16372-predict-the-weather.py	8	0	0	1	0	0
0-18-loss-simple-feature-extractors.py	13	0	0	0	1	1
0-2-data-science-tool-box.py	2	1	0	0	0	1
0-285-lb-average-of-3xgboost-lightgbm-nn.py	0	0	0	0	0	0
0-29-public-lb-score-beginner-nlp-tutorial.py	22	0	0	1	1	1
0-3-data-science-cleaning-data.py	2	1	0	0	0	1
0-31611-private-leader-board.py	2	0	0	1	0	1
0-335-log-loss-in-a-dozen-lines.py	0	0	0	0	0	0
0-49-publiclb-simple-blend-private-lb-rank-126th.py	0	0	0	0	0	0
0-573-lb-score-in-10-lines-of-code.py	0	0	0	0	0	0
0-6-lb.py	0	0	0	0	0	0
0-60-on-lb-w-max-mean-time-plus-trick.py	0	0	0	0	0	0

Table 5.2: Example of excessive imports files

The results of the track are shown in Table 5.3. The total count of the imports that were presented but never used is 157,394. Taking into account the total number of imports of 26,320 we see that more than 30% of the imports are used without any necessity.

NumPy library has the biggest count of excessive usage from the project's point

of view. This fact corresponds to the initial modules count, where we mentioned NumPy as the most used module. However, the result of *scikit-learn* shows that this module is often used without the required experience, and data scientists are just copying a block of code without considering if it is required. The maximum number of excessive imports per project is 90 (the project file is densenetimaug.py).

From the projects point of view, 94% of projects has at least one excessive import. The median of the excessive imports per project is 3.

Module name	Excessive count (projects)
scikit-learn	20,748
Keras	6,568
pandas	5,957
NumPy	23,938
Matplotlib	12,870

Table 5.3: Excessive imports of modules

In order to get a deeper understanding of the usage of the libraries, we decided to do further analysis on the usage of sub-modules. For each of the top modules we analysed counts of the sub-modules in *import* and *from import* statements. For each of the modules corresponding CSV file was created with the sub-modules imports counts. Table 5.4 presents the counts of the sub-modules imports per top module. From those results, we could see that *Scikit* library has most of the sub-modules imports. Even more, we could see that only 7.4% of total *Scikit* imports are direct imports. We see a similar situation with *Matplotlib* library, which has 10% of direct imports. Almost the same situation we could see with Keras library - 11%. However, we could see a completely different situation with NumPy and pandas modules. NumPy has 99% of direct imports, pandas - 98%. This means that data scientists typically import the library itself, not part of the library. To answer the question of why we have such a situation, we should check the library's structures. NumPy and pandas are

the libraries dedicated to data manipulation. Those libraries do not have a complex structure of sub-libraries. Those mainly contain classes that represent data structures and methods that are used for data manipulation. On the other hand, Matplotlib has more than 60 sub-libraries, and most of the common-used methods are in the sub-libraries.

Module name	Sub-modules imports count
scikit-learn	213,749
Keras	92,695
pandas	1,645
NumPy	530
Matplotlib	54,649

Table 5.4: Sub-modules imports per top module

As the result, we decided to extend modules usage analysis to cover sub-modules in Scikit, Keras, and Matplotlib. Based on the extract data, we discover that for Scikit and Keras we could specify Top-5 sub-modules that are mostly used. However, analysis of the Matplotlib sub-modules showed that only 90% of the sub-modules calls refers to one sub-module - matplotlib.pyplot. The results are presented in the Table 5.5.

Scikit		Keras		Matplotlib	
sub-module	count	sub-module	count	sub-module	count
metrics	51,201	layers	45,448	pyplot	48,916
model_selection	45,478	models	11,019	image	958
preprocessing	23,745	callbacks	7,817	patches	762
ensemble	19,360	optimezers	5,144	pylab	694
linear-model	19,048	preprocessing.image	3,946	colors	680

Table 5.5: Sub-modules count

5.4 RQ4: Calls usage analysis

The first conclusion that was made after the initial analysis - not all the calls we extracted are methods. The extracted data contains counts of the method calls and Python class calls. The quantitative results of the calls usage per module are presented in the Table 5.6.

Module name	Methods & classes count
scikit-learn	232,350
Keras	138,418
pandas	288,219
NumPy	437,391
Matplotlib	636,168
Total	**1,732,546**

Table 5.6: Methods count per modules

During the further analyses, we faced the situation that, similarly to the situation with the sub-modules, the usage of the calls is not homogeneous. For the pandas module, *read_csv* takes 40% of all calls. Another 25% was used by the DataFrame class. The total number of calls variations for pandas is 162. Nevertheless, 10 of the call variations are responsible for 93% of all calls.

For the Matplotlib and NumPy, the modules with the most amount of calls, we have from small to no amount of classes in the calls. Matplotlib has 99% of the methods in the calls.NumPy has only 0.4% of classes in the calls. For the Keras module, the situation is completely different. From the first (by usage frequency) twenty calls, only two are the methods; all others are classes.

Following the observations described earlier we could define two types of the libraries - method-oriented and class-oriented. Method-oriented library provides developers with the wide set of methods that could be used for any purpose. From the other hand, class-oriented library provides wide set of classes that developer can

instantiate. The methods provided by the class-oriented library usually are applied to the classes.

Table 5.7 presents top 10 calls per library together with the count for each call.

NumPy		pandas		Scikit		keras		Matplot	
Call	count	Call	count	Call	count	Call	count	Call	count
tanh	40460	read_csv	106445	train_test_split	24436	Dense	22682	show	94414
array	37718	DataFrame	69762	accuracy_score	14936	Conv2D	14628	figure	74041
where	28517	concat	24737	confusion_matrix	12327	Dropout	12239	title	72842
zeros	18655	get_dummies	13888	RandomForestClassifier	9622	BatchNormalization	7630	ylabel	48300
arange	18403	merge	11771	LogisticRegression	8382	Sequential	7011	xlabel	48050
mean	17392	to_datetime	10541	mean_squared_error	8009	MaxPooling2D	5249	plot	47029
sqrt	10840	Series	10190	classification_report	7788	Activation	5101	subplots	39755
sum	9101	set_option	5107	cross_val_score	6898	Input	4191	subplot	25832
nan	8749	to_numeric	4075	GridSearchCV	6278	Flatten	4065	legend	24728
sin	8420	crosstab	3695	StandardScaler	6198	Model	3814	xticks	19818

Table 5.7: Top-10 calls count per library

5.5 Similarity analysis

As we found out and described in chapter 3.2, more than 40% of the projects are solving the top ten competition. That is why we assume that it would be using full to investigate how similar the code is. In order to find this out, we decided to choose one competition and select five projects that are solving the problem in this competition. Our natural hypothesis was that the projects dedicated to the same project would have similar code. We took five projects (Table 5.8) that are using the data for the same competition - Titanic: Machine Learning from Disaster [46]. This competition assumes that in addition to luck influence, some of the people that were on the sinking Titanic ship had more chances to survive than others. The task of the competition is to produce a predictive model that would help to answer what type of people would likely survive. The data set of the task contains passenger data. This competition has the highest amount (6,058) of projects, dedicated only to one competition.

During similarity analysis we compared following:Libraries usage, Libraries excessive imports and Calls per library per project.

File name	File size (bytes)	LOC
titanic-catboost.py	34,119	442
titanic-challenge-survivability-prediction.py	28,021	416
titanic-classification-comprehensive-modeling.py	25,051	337
titanic-challenge-crisp-dm.py	52,132	290
titanic-classification-problem-beginner.py	25,960	279

Table 5.8: Files used for similarity analysis

The first comparison was done straightforward by comparing the list of libraries that were imported in each project. This comparison showed that four out of five projects used the same set of libraries - NumPy, pandas, scikit-learn, and Matplotlib. The fifth project (titanic-challenge-survivability-prediction) did not use 2 libraries - Keras and Matplotlib. However, the picture was not complete as we were not sure if the libraries were imported and used.

We analyzed how many libraries were excessively imported. The results are presented in the Table 5.9. Four of the projects have almost the same count of excessive imports. Even more, all projects have excessive imports in the scikit library. This is the expected result, as we described earlier that scikit has the most amount of excessive imports comparing with other libraries. We checked excessive imports for each project in detail, and all projects have a high amount of scikit imports; some of those imports are excessive, but the library is used.

File name	Excessive count total	NumPy	pandas	scikit	keras	plot
titanic-catboost.py	6	0	1	1	0	0
titanic-challenge-crisp-dm.py	2	0	0	1	0	0
titanic-challenge-survivability-prediction.py	3	0	0	1	0	1
titanic-classification-comprehensive-modeling.py	1	0	0	1	0	0
titanic-classification-problem-beginner.py	2	0	0	1	0	0

Table 5.9: Excessive imports comparison

Next, the files were compared by the calls they were using in each library. The data is presented in Table 5.10. It shows that none of the methods is presented in all projects. However, we could clearly see that some set of methods is commonly used.

For example, we see that read_csv and DataFrame from pandas library is used in four out of five projects. Even more, the table clearly demonstrates the difference between the libraries. NumPy, pandas and Matplotlib are method-based, scikit is class based library.

File name	Calls			
	NumPy	pandas	scikit	Matplot
titanic-catboost.py	log sqrt where seed	DataFrame read_pickle concat	accuracy_score PowerTransformer StandardScaler train_test_split	title figure subplots xlabel rcParams ylabel use
titanic-challenge-survivability-prediction.py	bool triu_indices_from zeros_like filterwarnings	get_dummies read_csv cut merge concat DataFrame	DecisionTree, Classifier, SVC ExtraTreesClassifier, ElasticNet Pipeline,BaggingClassifier,VotingClassifier MinMaxScaler,AdaBoostClassifier MLPClassifier,RobustScaler RandomForestClassifier,Normalizer StratifiedKFold,GradientBoostingClassifier LogisticRegression,GaussianProcessClassifier cross_val_score,LinearDiscriminantAnalysis KNeighborsClassifier,StandardScaler	
titanic-classification-comprehensive-modeling.py	round logspace	DataFrame concat cut read_csv get_dummies	cross_val_score,DecisionTreeClassifier KNeighborsClassifier, LogisticRegression RandomForestClassifier SVC BaggingClassifier GridSearchCV LabelEncoder	figure title show
titanic-challenge-crisp-dm.p	where float nan	Series cut concat read_csv qcut get_dummies	roc_auc_score MinMaxScaler RandomForestClassifier LogisticRegression AdaBoostClassifier train_test_split	subplot figure
titanic-classification-problem-beginner.py	round arange mean	get_dummies read_csv concat DataFrane isnull	accuracy_score,classification_report confusion_matrix,LogisticRegression RandomForestClassifier,KNeighborsClassifier VotingClassifier,GridSearchCV StandardScaler,train_test_split	subplots tight_layout title figure xlabel plot ylabel

Table 5.10: Calls usage per library

The similarity analysis proves to us that despite the fact that the projects are dedicated to the same problem solving, we cannot state that the set of methods is the same for all projects. However, some patterns could be seen and could possibly be used for further research.

5.6 Threats to Validity

There are several threats to the validity of the results discussed in this section. The first is that the code analyzed in this study may not be representative of the code used as part of Kaggle competitions (internal validity), while the second is that the code analyzed in this study may not be representative of the machine learning code that developers write in general (external validity).

As mitigation against the first concern, code has been selected based on the data released as part of the Meta Kaggle dataset, with no additional filtering applied. Assuming that the data provided as part of Meta Kaggle is itself representative of the code submitted to Kaggle, the analysis will have examined a representative collection of Python machine learning code submitted to Kaggle.

As mitigation against the second concern, as discussed in Section 3, code has been selected across a wide variety of different Kaggle challenges and data sets, from a wide variety of authors. This should ensure the analysis is not just identifying libraries that are appropriate for a specific challenge or a specific data domain, or identifying patterns of use from a limited collection of authors.

Chapter 6

Conclusions and Future Work

In this chapter, we first summarize the results of our study in Section 6.1. We then describe extensions to this work in Section 6.2.

6.1 Conclusions

The complexity of the ML APIs learning process is due to variety of the libraries, classes and methods that could be used to solve different data scientific tasks. a recommender system, that would advise students which libraries and methods they could use in different situations could be the great help. In order to create a recommender system it is necessary to understand how the ML APIs are used.

To understand more about how ML APIs are used in Python, we set out to answer the following questions:

- **RQ1:** What are the main programming languages used for data science applications?
- **RQ2:** What are the main machine learning libraries used in Python machine learning code?
- **RQ3:** In each of these libraries, what are the most used modules?
- **RQ4:** For each of the libraries and modules, what are the most common API calls?

To answer the research questions we produced the data corpus based on the Python source code files. Those files were extracted from the Kaggle competition platform. The files were written by different independent data scientists and were downloaded and processed during our research. All answers for the research questions are based on this corpus.

In order to answer the RQ1 we studied the questionnaire made by Kaggle and analyzed the PL used in the source code files. Our results demonstrates, that Python should be the target language for the recommender system, dedicated to the ML API. Even more, we see that there are different types of the approaches to the writing of the Python code: Python code and Jupyter notebooks. Jupyter notebooks are used by data scientists and are more and more involved in the educational process [13, 9]. In our research we had more than 90% of explored source code written using Jupyter. The block execution approach used in Jupyter Notebooks together with possibility include comments, graphs, video etc. make this type of Python programming very attractive. That is why it could be reasonable to focus the future recommender system on Jupyter notebooks.

While answering the RQ2, we have extracted the libraries that were imported in each file. We have defined the list with Top 5 used Machine Learning related libraries: NumPy, pandas, Scikit-learn, Keras and Matplotlib. Even more, we have found out that more than 30% imports of the libraries are stated but never used.

Next, we have distilled the modules and calculated the frequency of their usage. Our research shows that the libraries are not homogeneous from the modules point of view. NumPy and pandas are not module-oriented libraries. The usage of modules is strong for Scikit-learn, Keras and Matplotlib libraries. The top modules for those libraries are: metrics for Scikit, layers for Keras, and pyplot for Matplotlib. The list with top 5 used module per those 3 libraries is presented in Chapter 5.

Finally, we have answered the RQ4 and received the list of calls that are mostly used per each library. The top calls for the top 5 libraries are: tahn for NumPy, read_csv for pandas, train_test_split for Scikit-learn, Dense class for Keras and show for Matplotlib. Thetop 10 calls per library could be found in Table 5.7.We have extracted 1,732,546 calls for the top modules. We have found that not all calls are methods. That is why we separate libraries to class-oriented and methods-oriented groups. We also stated that the usage of the calls is not even and some methods (e.g., read_csv in pandas) could be in almost half of the calls.

During the data analysis, we have compared the sample of five projects that were dedicated to the same competition. As a result, we found out that it is not possible to state that the projects targeted the same problem use the same set of calls. However, we see that it should possible to find some patterns of the ML API usage in those projects.

6.2 Future work

Based on the results presented above, we believe the following would be interesting questions for future research.

- *Try to improve corpus.* Search for the possibility to get the complete database with competitions and kernels.
- *Find the patterns approach.* Our research demonstrates that some patterns could be found while analyzing the calls usage. We suppose that it should be possible to look for a larger usage patterns, involving larger parts of code. Those patterns could involve other expressions as well as multiple calls.
- *Implement recommender system.* Our current work together with the patterns approach mentioned above that could be used to provide API and code snippet recommendations.
- *Study the possibility to integrate the system in the Jupyter Notebook editor.* It would be a good idea take into the account the check for excessive imports.

- *Quality of ML APIs usage study.* During our research, we noticed that, from our point of view, there are some issues with the code quality, especially in the source code written with Jupyter Notebooks. It could be useful to compare programming skills and experience of the developers who uses Jupyter Notebook versus developers who directly creates Python files. Even more, it would be useful to look for difference in patterns between those two approaches. Finally, it would be useful to find a way to refactor a code to improve the quality of ML APIs usage in notebooks.

- *Expand existing approach to support additional corpuses.* In our research, we focused on a Kaggle-based corpus of data. It could be interesting to extend the approach to analyze not only solutions presented by one file but also the ones presented by projects consisting of potentially complex directory structures with multiple files. It may also be useful to look beyond Kaggle to find potential projects for an enriched corpus. This could include projects referenced in web pages and academic publications related to machine learning.

BIBLIOGRAPHY

[1] API Directory. `https://www.programmableweb.com/apis/directory`.

[2] APIs Growth Rate. `https://www.programmableweb.com/news/apis-show-faster-growth-rate-2019-previous-years/research/2019/07/17`.

[3] Asaduzzaman, M., Roy, C. K., Schneider, K. A., and Hou, D. Context-Sensitive Code Completion Tool for Better API Usability. In *2014 IEEE International Conference on Software Maintenance and Evolution* (2014), IEEE, pp. 621–624.

[4] Bao, L., Xing, Z., Wang, X., and Zhou, B. Tracking and Analyzing Cross-Cutting Activities in Developers' Daily Work (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2015), IEEE, pp. 277–282.

[5] Bhoopchand, A., Rocktäschel, T., Barr, E., and Riedel, S. Learning Python Code Suggestion with a Sparse Pointer Network. *arXiv preprint arXiv:1611.08307* (2016).

[6] Bruch, M., Bodden, E., Monperrus, M., and Mezini, M. IDE 2.0: collective intelligence in software development. In *Proceedings of the FSE/SDP workshop on Future of software engineering research* (2010), pp. 53–58.

[7] Bruch, M., Monperrus, M., and Mezini, M. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering* (2009), pp. 213–222.

[8] Date, C. J., and Codd, E. F. The relational and network approaches: Comparison of the application programming interfaces. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control: Data models: Data-structure-set versus relational* (1975), pp. 83–113.

[9] DePratti, R. Using Jupyter notebooks in a big data programming course. *Journal of Computing Sciences in Colleges 34*, 6 (2019), 157–159.

[10] D'Souza, A. R., Yang, D., and Lopes, C. V. Collective intelligence for smarter API recommendations in Python. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2016), IEEE, pp. 51–60.

[11] Ellis, B., Stylos, J., and Myers, B. The factory pattern in API design: A usability evaluation. In *29th International Conference on Software Engineering (ICSE'07)* (2007), IEEE, pp. 302–312.

[12] The state of the octoverse: Machine learning. `https://github.blog/2019-01-24-the-state-of-the-octoverse-machine-learning/`.

[13] Glick, B., and Mache, J. Using Jupyter notebooks to learn high-performance computing. *Journal of Computing Sciences in Colleges 34*, 1 (2018), 180–188.

[14] Hellendoorn, V. J., and Devanbu, P. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), pp. 763–773.

[15] Hindle, A., Barr, E. T., Su, Z., Gabel, M., and Devanbu, P. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 837–847.

[16] Hora, A. APISonar: Mining API usage examples. *Software: Practice and Experience* (2020).

[17] Project Jupyter. `https://jupyter.org/`.

[18] Kaggle. `https://www.kaggle.com/`.

[19] 2018 Kaggle ML & DS Survey. `https://www.kaggle.com/kaggle/kaggle-survey-2018`.

[20] Karampatsis, R.-M., Babii, H., Robbes, R., Sutton, C., and Janes, A. Big Code!= Big Vocabulary: Open-Vocabulary Models for Source Code. *arXiv preprint arXiv:2003.07914* (2020).

[21] Li, J., Wang, Y., Lyu, M. R., and King, I. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573* (2017).

[22] Matplotlib homepage. `https://matplotlib.org/`.

[23] Meta Kaggle. `https://www.kaggle.com/kaggle/meta-kaggle`.

[24] Myers, B. A., and Stylos, J. Improving API usability. *Communications of the ACM 59*, 6 (2016), 62–69.

[25] Nguyen, H. A., Dyer, R., Nguyen, T. N., and Rajan, H. Mining preconditions of APIs in large-scale code corpus. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), pp. 166–177.

[26] Nguyen, T. T., Nguyen, A. T., Nguyen, H. A., and Nguyen, T. N. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013), pp. 532–542.

[27] Niu, H., Keivanloo, I., and Zou, Y. API usage pattern recommendation for software development. *Journal of Systems and Software 129* (2017), 127–139.

[28] Numpy homepage. `https://numpy.org/`.

[29] pandas homepage. `https://pandas.pydata.org/`.

[30] Python 3.8.6rc1 documentation - Import. `https://docs.python.org/3/reference/simple_stmts.html#import`.

[31] Raschka, S., Patterson, J., and Nolet, C. Machine Learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information 11*, 4 (2020), 193.

[32] Robbes, R., and Lanza, M. How program history can improve code completion. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering* (2008), IEEE, pp. 317–326.

[33] Robillard, M. P. What makes APIs hard to learn? Answers from developers. *IEEE software 26*, 6 (2009), 27–34.

[34] Robillard, M. P., and Deline, R. A field study of API learning obstacles. *Empirical Software Engineering 16*, 6 (2011), 703–732.

[35] Sawant, A. A., and Bacchelli, A. fine-GRAPE: fine-grained APi usage extractor–an approach and dataset to investigate API usage. *Empirical Software Engineering 22*, 3 (2017), 1348–1371.

[36] scikit-learn homepage. `https://scikit-learn.org/stable/`.

[37] Scipy homepage. `https://www.scipy.org/`.

[38] Srinath, K. Python–The Fastest Growing Programming Language. *International Research Journal of Engineering and Technology (IRJET) 4*, 12 (2017), 354–357.

[39] Stylos, J., and Clarke, S. Usability implications of requiring parameters in objects' constructors. In *29th International Conference on Software Engineering (ICSE'07)* (2007), IEEE, pp. 529–539.

[40] Stylos, J., and Myers, B. A. Mica: A Web-search tool for finding API components and examples. In *Visual Languages and Human-Centric Computing (VL/HCC'06)* (2006), IEEE, pp. 195–202.

[41] Svyatkovskiy, A., FU, S., Sundaresan, N., and Zhao, Y. Deep Learning Enhanced Code Completion System, Aug. 6 2020. US Patent App. 16/377,789.

[42] Svyatkovskiy, A., Zhao, Y., Fu, S., and Sundaresan, N. Pythia: AI-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2019), pp. 2727–2735.

[43] Svyatkovskoy, A., Lee, S., Hadjitofi, A., Riechert, M., Franco, J., and Allamanis, M. Fast and Memory-Efficient Neural Code Completion. *arXiv preprint arXiv:2004.13651* (2020).

[44] TensorFlow. `https://www.tensorflow.org/`.

[45] Theano Web site. `http://deeplearning.net/software/theano/`.

[46] Titanic: Machine Learning from Disaster — Kaggle. `https://www.kaggle.com/c/titanic`.

[47] Tosi, S. *Matplotlib for Python developers*. Packt Publishing Ltd, 2009.

[48] Tu, Z., Su, Z., and Devanbu, P. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), pp. 269–280.

[49] Xia, X., Bao, L., Lo, D., Kochhar, P. S., Hassan, A. E., and Xing, Z. What do developers search for on the web? *Empirical Software Engineering 22*, 6 (2017), 3149–3185.

[50] Xu, C., Sun, X., Li, B., Lu, X., and Guo, H. MULAPI: Improving API method recommendation with API usage location. *Journal of Systems and Software 142* (2018), 195–205.

[51] Zhong, H., and Mei, H. An empirical study on API usages. *IEEE Transactions on Software Engineering 45*, 4 (2017), 319–334.

[52] Zhong, H., Xie, T., Zhang, L., Pei, J., and Mei, H. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming* (2009), Springer, pp. 318–343.