University of Zurich[UZH]

Automatic Annotation of Data Science Notebooks

A Machine Learning Approach

MSc Thesis March 11, 2019

Dhivyabharathi Ramasamy
of Coimbatore TN, India

Student-ID: 14-924-823
img_dhivyabharathi.ramasamy@uzh.ch

Advisors: **Cristina Sarasua**

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

First, I would like to thank Prof. Abraham Bernstein for giving me the opportunity, after the master basis module and the master's project, to also write my thesis at the Dynamic and Distributed Information System Group of the University of Zurich. I would also like to thank him for his support, domain expertise and valuable insights that he offered during my thesis.

I would like to express my sincerest gratitude to my supervisor Cristina Sarasua for her guidance and valuable insights. I have been lucky to have a supervisor who cared so much about my work, provided constant support and appreciation throughout my thesis.

Last but not least, I would like to thank my family and friends for their unfailing support and continuous encouragement throughout my studies. This accomplishment would not have been possible without them. Thank you.

Zusammenfassung

Daten-Wissenschafts-Notebooks sind die Notebooks, die für Datenwissenschaftstätigkeiten wie Erforschung, Zusammenarbeit und Visualisierung entwickelt werden. Traditionsgemäss verwendet, wie ein instrument, zum von reproduzierbaren Ergebnissen und von Dokumentation der Forschung bereitzustellen, sie in den letzten Jahren wegen der enormen Zugkraft im Bereich des maschinellen Lernens bekannt geworden. Wechselwirkende Notebooks sowie Jupyter, Zeppelin, und Kaggle sind etwas von dem Primärplattformleutegebrauch für die Implementierung einer Datenwissenschaftsaufgabe. Die Notebooks, benutzt von den Datenwissenschaftlern, um ihre Datenwissenschaftsaufgaben einzuführen, sind eine wichtige Quelle von den Daten für das Verständnis und das Analysieren von Datenwissenschaftsrohrleitungen in der Praxis eingeführt geworden. Jede Datenwissenschaftsrohrleitung enthält viele Datenwissenschaftstätigkeiten und zwecks sie zu analysieren, ist es notwendig, zu identifizieren, wo in einem gegebenen Notebooks jede Datenwissenschaftstätigkeit stattfindet. Die Datenwissenschaftstätigkeiten in den notebooks durch Experten zu beschriften ist ein Zeit raubender und teurer Prozess. In dieser Master-Arbeit versuchen ich, die Datenwissenschaftstätigkeit/-tätigkeiten jedem *Zelle* der Datenwissenschafts Notebooks unter Verwendung überwachtes maschinelles Lernen zu klassifizieren und zuzuweisen. Ich haben einen Satz allgemeine hochrangige Datenwissenschaftstätigkeiten als Aufkleber identifiziert und jedes Notebooks *Zelle* die Aufkleber zuweisen, die auf der Datenwissenschaftstätigkeit basieren, die sie durchführen. Mehrfache Datenwissenschafts-Tätigkeitsaufkleber sind zu jedem *Zelle* wegen der unterschiedlichen Kodierungsart der notebook benutzer, der Überschneidungstätigkeiten, des etc. erlaubt worden. Ein Anmerkungsexperiment war entworfen und geleitet, um expert/s beschriftete zu erhalten Daten und ein Satz von 100 Experte angemerkten jupyter Notebooks wird als Datensatz in den Experimenten benutzt. *Python* sind Klassen entwickelt worden, um verschiedene Merkmale aus den jupyter Notebooks für die Klassifikationsaufgabe zu extrahieren. Mehrfachverbindungsstelle überwachte Klassifikatoren (KNearest Neighbors, Support Vector Machine, Multi-layer Perceptron, Gradient Boosting, Random Forest, Decision Tree, Naive Bayes, Logistic Regression) sind unter Verwendung der Klassifikationsmethoden Singlelabel und Multilabel für die Klassifikationsaufgabe ausgewertet worden. Logistic Regression klassifikator unter Verwendung Multilabel-Klassifikation hat im Vergleich zur Singlelabel Classification eine höhere Präzision. Die Forschung zeigt, dass Ensemble methoden und Logistic Regression für die Klassifikation des Quellcodes geschrieben in Notebooks passender sind. Die Bedeutung des Merkmale,

die in den Forschungsfragen besprochen werden, stellen Einblicke in die informativen Merkmale für Codeklassifikation zur Verfügung. Der Vergleich der zwei Klassifikationsparadigmen und der besseren Leistung von Multilabel-Klassifikation im Hinblick auf Präzision führt zu die Schlussfolgerung, dass Datenwissenschaftsrohrleitungen, wie in den Notebooks gefunden nicht immer sequenziell sind und in hohem Grade die meisten Zeiten überschneiden, sich im Vergleich zum theoretischen Design von Datenwissenschaftsrohrleitungen. Ich haben auch eine Ontologie für Notebooks und die Datenwissenschaftstätigkeiten entwickelt und verwenden diese, um die Anmerkungen in der semantischen Netzart zur Verfügung zu stellen, im Resource Description Framework (RDF)[1] format gespeichert für weitere Analyse. Darüber hinaus haben ich auch Ergebnisse der explorativen Datenanalyse und der Leistung der unüberwachten Klassifikation auf dem Datensatz produziert und diskutiert. Eine Analyse der inter-annotator Vereinbarung wird auch diskutiert. Es ist wichtig, zu erwähnen, dass die Merkmals, die unter Verwendung dem Systems erzeugt werden, in den Analysen auch benutzt werden können, die in andere Kontexten eingestellt werden.

[1] *https://en.wikipedia.org/wiki/Resource_Description_Framework*

Abstract

Data Science Notebooks are notebooks developed for data science activities like exploration, collaboration, and visualization. Traditionally used as a tool to provide reproducible results and documenting the research, they have become prominent in the last few years due to the enormous traction in Machine learning field. Interactive notebooks like Jupyter, Zeppelin, and Kaggle are some of the primary platforms people use for implementing a data science task. Notebooks, used by data scientists to implement their data science tasks, have become an important source of data for understanding and analysing data science pipelines implemented in practice. Each data science pipeline contains many data science activities and in order to analyse them, it is necessary to identify where in a given notebook each data science activity takes place. Labelling the data science activities in the data science notebooks by experts is a time consuming and expensive process. In this master thesis, I attempt to automatically classify and assign the data science activity/activities to each *cell* of the data science notebooks using supervised machine learning. I have identified a set of common high-level data science activities as labels and assign each notebook *cell* the labels based on the data science activity they perform. Multiple data science activity labels have been allowed to each *cell* due to different coding style of the notebook users, overlapping activities, etc. An annotation experiment was designed and conducted to get expert/s labelled data and a set of 100 expert-annotated jupyter notebooks is used as a dataset in the experiments. *Python* classes have been developed in order to extract various features from the jupyter notebooks for the classification task. Multiple supervised classifiers (KNearest Neighbors, Support Vector Machines, Multi-layer Perceptron, Gradient Boosting, Random Forest, Decision Tree, Naive Bayes, Logistic Regression) have been evaluated using both Singlelabel and Multilabel Classification methods for the classification task. Logistic Regression classifier using Multilabel Classification has a higher precision compared to Singlelabel Classification. The research shows that ensemble methods and logistic regression are more suitable for classification of source code written in notebooks. Features importances discussed in the research questions provide insights into the informatory features for code classification. The comparison of the two classification paradigms and better performance of Multilabel Classification in terms of precision leads to the conclusion that data science pipelines as found in notebooks are not always sequential and are highly overlapping most of the times compared to the theoretical design of data science pipelines. I have also developed an ontology for notebooks and the data science

activities and use the same to provide the annotations in semantic web style serialized in Resource Description Framework (RDF)[2] format for further analysis. In addition, I have produced and discussed the results of exploratory data analysis and the performance of unsupervised classification on the dataset. An analysis of inter-annotator agreement is also discussed. It is important to mention that the features generated using the system can also be used in analyses set in other contexts.

[2] *https://en.wikipedia.org/wiki/Resource_Description_Framework*

Table of Contents

List of Figures

List of Tables

1

Introduction

1.1 Motivation

Data science is an interdisciplinary field that uses scientific methods, processes, algorithms, and systems to extract knowledge and insights from data in various forms, both structured and unstructured [Dhar, 2013]. Given the enormous amount of data generated and data science techniques developed every day [Hey et al., 2009] [Gordon Bell, 2009], Data science as a field continues to gain increasing importance. It should also be noted that the data science methods are developed and used by not only computer scientists, mathematicians, and statisticians but also people from various non-STEM fields for achieving their various needs. While data science techniques have been used to solve a wide range of problems from automation to healthcare, there are scarce examples of works studying the way data science pipelines/processes are implemented [Leek, 2013]. To be able to better design methodologies and frameworks that enable high-quality data science for evolving needs, it is necessary to understand the way data science is designed, implemented and reported by data science practitioners which provides the motivation for the master thesis. Ultimately, the goal is to guarantee high-quality design and execution of data science processes to every data scientist.

This thesis is a first step towards addressing this goal, as I investigate methods to automatically annotate data science notebooks[1], labelling parts of the notebooks that refer to each of the steps in the data science process.

1.2 Research Questions

Goal. The main goal of this master thesis is to design and implement a method that automatically classifies and annotates the parts of the data science notebook into data science steps/activities. Given a set of notebooks containing multimodal data (code, natural language like comments or markdowns, media like graphs or animations etc.), the task is to automatically classify, label and annotate the parts of each of the them according to the Data Science activity they carry out using state of the art machine learning

[1]Notebook, notebook, and Jupyter notebook are used interchangeably

methods [Pustejovsky and Stubbs, 2012]. The research questions that we address are precisely:

1. What features in the notebook are more informatory to the automatic classification of data science notebooks?
2. Do non-code features like markdown/comments in notebooks improve classification accuracy?
3. Are import statements along with their library functions sufficient to classify the code according to their data science activities?
4. Do popular[2] coders produce notebooks that are easier to classify/have higher classification accuracy?

1.3 Contribution of the Thesis

The contributions of this master thesis are:

1. A method to automatically classify and annotate the parts of the data science notebooks according to the data science activity it performs.
2. An evaluation of various classification methods and techniques to automatically classify the parts of the data science notebooks.
3. An ontology to annotate data science activities in the notebooks.
4. A data set containing RDF annotations about the notebooks and the data science activities in them.

1.4 Thesis Structure

The rest of the thesis is structured as follows: Chapter 2 discusses the related work and background knowledge in the field. In Chapter 3, the concepts discussed throughout the thesis are introduced. Chapter 4 discusses the system and its components in detail. The system is evaluated using the dataset presented in Chapter 5. Chapter 5 also presents the methodology and an extensive analysis of the data which is followed by the results of the classification task using supervised machine learning techniques. The chapter also presents the output of the classification task using RDF annotations. Chapter 6 discusses the results and its insights further in detail. Rest of the chapters conclude the thesis through Conclusions and Future Work.

[2]Popularity of a user is identified using a number of features: forks, stars/upvoters, watchers count.

2

Related Work

Classification of source code for various purposes (identification of programming language, authorship, syntax classification etc) has been explored for many years now while scarce examples of studies that classify source code from notebooks exist. Only recently, analysis of data science notebooks has gained attention and in this chapter, I will discuss some of the existing works relevant to classification of source code in the context of this thesis. I also discuss in this chapter, existing ontologies representing any information related to the data science process and activities that is relevant for the annotation task.

2.1 Analysis of Data Science Notebooks

So far, there has not been any published papers[1] that classify notebook parts according to the data science activity.

2.2 Code Classification

More research has been focused on using machine learning techniques for classifying source code but in different contexts. [Barstad et al., 2014] uses static code analysis and machine learning for predicting the code quality. K-Nearest Neighbour (KNN), Naive Bayes (NB) and Decision Tree (DTree) are compared as classifiers with Naive Bayes performing the best for predicting badly and well-written code. [Zevin and Holzem, 2017] uses max-entropy classifier for programming language prediction. Other papers discussing classification methods for source code analysis in different contexts include [Knab et al., 2006] which focuses on defect prediction in source code using Decision Tree and [Ugurel et al., 2002] which discusses the multi-class classification of source code by category, and programming language using Support Vector Machines (SVM). Some of the other papers discussing text classification in general include [Burges, 1998] using SVM and [Cavnar and Trenkle, 1994] which focuses on n-grams based text classification.

[1] to the best of my knowledge

In [Binkley, 2007], authors have published a summary on source code analysis and discuss extensively the research that has been done previously and also on what challenges lies ahead. I focus on the review about the first of three components (the parser, the internal representation, and the analysis of this representation) in source code analysis, parsing, which is relevant for our machine learning approach. "Parsing is the necessary evil of most source-code analysis," says [Binkley, 2007] and suggests that source code analysis should move from lexical and to semantics which is something I have left for the future work.

User Style Features. [Pellin, 2006] focused on prediction of code authorship says that user style of a coder is not captured because of the presence of auto-filling and other automatic code writing features. Instead of flat tree representation with term counts, the authors represent features using syntactical tree structures. As notebooks do not support a structured coding environment like a typical IDE, user style plays an important role in the way a function is implemented or a code block is written. The authors also represent each data point at a function-level granularity to represent which is not always typical in notebooks. The function-level granularity also increases the documents in the corpora and the authors suspect that programming patterns exist in methods.

Code Metrics Features. Halstead and McCabe Cyclomatic Complexity are used as features in [Barstad et al., 2014] for predicting source code quality combining rule-based static code analysis and machine learning.

Syntax based Features. In [Zevin and Holzem, 2017], authors use syntactical features to predict the programming language of the source code using Maximum Entropy Classifier. The authors replace alpha, numeric and punctuation sequences with constants since each such sequences represents token of different type and n-grams (uni, bi, tri) produced using WEKA[2] for grammar structures of programming language. [Ugurel et al., 2002] uses words from code (including header file names) and comments. The authors have used words, bigrams and lexical phrases extracted from comments and README files for classifying source code by topic and programming language.

Sequence Capturing Features. Discovering sequences of labels or subsequences in a set of sequences is of interest to us and helpful in identifying how data science pipelines are designed. While sequential pattern mining for source code has not gathered attention, few papers have been published that do consider the sequential nature of the lines of code in the context of feature location. In [Bacchelli et al., 2012], for the purpose of content classification of development mails, the authors use features “@@-lineBefore”, and “@@-lineAfter” to consider the lines before and after to recognize the structure of patch or stack trace content.

[2] *https://www.cs.waikato.ac.nz/ml/weka/*

2.3 Ontology modelling

There have been many ontologies created in the context of the semantic web, however, there are only a few focusing on the representation of research and data analysis process. Currently, no standard ontology for representing data science pipeline or notebooks information is currently in existence. An ontology for data science terms and activities[3] has been developed by a team from IBM Research AI[4] and Stanford University Statistics[5]. While this provides a vocabulary for many activities that is a part of a data science task, it does not include the concepts of a data science pipeline.

Another ontology, The Workflow Motif Ontology[6] that focuses on data related operation in a scientific workflow has been developed by a team from Ontology Engineering Group, Universidad Politécnica de Madrid, Spain, University of Southern California, USA and University of Manchester, UK, based on the Taverna [Missier et al., 2010] and Wings [Gil et al., 2011] workflows, i.e., the workflow of data in a general experiment. What is required for our task is an ontology that sits between the two ontologies discussed above with workflow ontology being a superclass and data science ontology being subclasses (see Section 4.1.2).

[3] *https://www.datascienceontology.org/about*
[4] *https://www.research.ibm.com/artificial-intelligence/*
[5] *https://statistics.stanford.edu/*
[6] *http://vocab.linkeddata.es/motifs/*

3

Preliminaries

This chapter introduces the concepts of a notebook, its structure, and characteristics. I also discuss in this chapter, different data science pipelines that are in existence.

3.1 A Notebook

Notebooks are documents produced by the Jupyter Notebook App and is of file type .ipynb. They contain all the content from a Jupyter Notebook Web application session including "both computer code (e.g. python) and rich text elements (paragraph, equations, figures, links, etc.). Notebook documents are both human-readable documents containing the analysis description and the results (figures, tables, etc.) as well as executable documents which can be run to perform data analysis"[1]. Jupyter notebook documents are stored in JSON plain text format and can be shared, and version-controlled. Each notebook is composed of a sequence of cells. Each cell is a multiline text input field which can be of type: code, markdown or raw. Notebooks also contain output cells which display the results from the execution of a code cell. Figure 3.1[2] illustrates a simple notebook and its parts.

3.1.1 JSON Structure of a Notebook document

Jupyter notebook files are simple JSON documents. They contain text, source code, rich media output, and metadata. A notebook at the highest level is a dictionary with the following keys[3]:

1. metadata (dict): contains arbitrary JSONable information about your notebook, cell, or output. Metadata used in this project are: kernel_language (notebook)

2. nbformat (int): declares notebook format

[1] *https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html#notebook-document*

[2] Figure adapted from *https://nbviewer.jupyter.org/github/ipython/ipython/blob/6.x/examples/IPythonKernel/SymPy.ipynb*

[3] *https://nbformat.readthedocs.io/en/latest/*

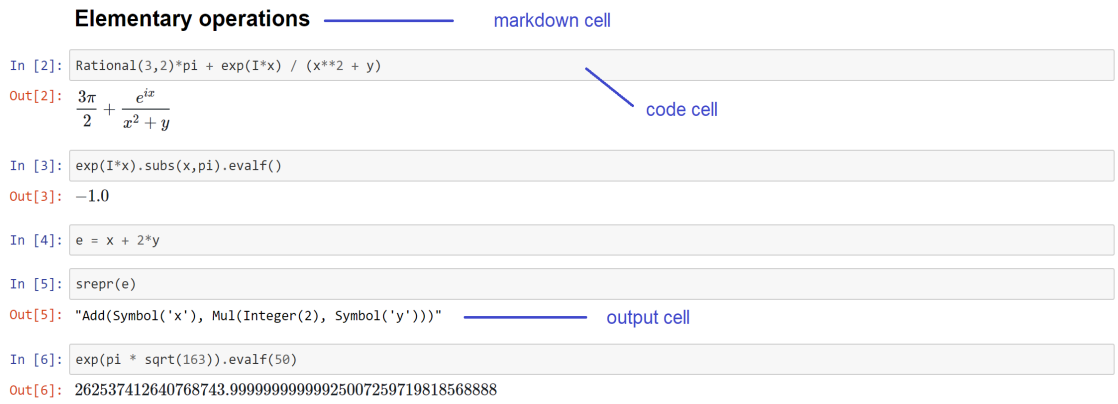

Figure 3.1: Basic structure of a notebook

```
{
  "cell_type" : "type",
  "metadata" : {},
  "source" : "single string or [list, of, strings]",
}
```

Figure 3.2: Basic structure of a notebook cell

3. nbformat_minor (int): declares notebook format

4. cells (list): Cell is a segment of the document encapsulating code and text. All the cells have a basic structure as shown in Figure 3.2[4]

3.1.2 Types of Cells

There are different types of cells in the notebook and they are described below.

1. Markdown: Markdown cells are used for body-text and contain markdown. Markdown is a lightweight markup language with plain text formatting syntax and is a superset of HTML. In Jupyter notebooks, plain text is added in the cell of type markdown. Markdown cell allows rendering of plain text, GitHub flavoured markdown or LaTeX and also allows embedding code.

2. Code: Code cells contain lines of code which are the primary content of a notebook. They can also contain comments. Code is written in the language associated with the kernel and on execution produces a list of outputs which are displayed in Output cell. Code cells may be denoted with execution_count of type int or null.

[4]Figure adapted from *https://jupyter-notebook.readthedocs.io/en/stable/notebook.html*

Figure 3.3: An example of stream output in notebooks

3. Raw NBConvert: Raw cells contain content that will not be rendered by notebook authoring environment and is not modified in nbconvert output (e.g. LaTeX).

4. Output: Output cells display output from the execution of a code cell. Output can be of various types: stream output, display_data, execute_result or error and is indicated by output_type.

3.1.3 Output types

1. stream output: stream output (stdout or stderr (see Figure 3.3[5]))

2. display_data: rich display output data with mime-type key.

3. execute_results: contains execute count and results of an execution.

4. error: traceback of a failed execution.

3.1.4 Cell attachments

Markdown and raw cells can have a number of attachments. The attachments can be referenced in the markdown content of a cell and are typically inline images[6].

3.2 Data Science Pipeline and Steps

In order to annotate notebooks in terms of the steps/activities in data science, it is important to first identify what these steps are. Different sources identify a different set of activities for a data science pipeline. Data science pipeline is a series of data-related activities connected in a sequential manner to go from obtaining the data to

[5] Figure adapted from *https://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/RunningCode.html*

[6] *https://nbformat.readthedocs.io/en/latest/format_description.html#cell-attachments*

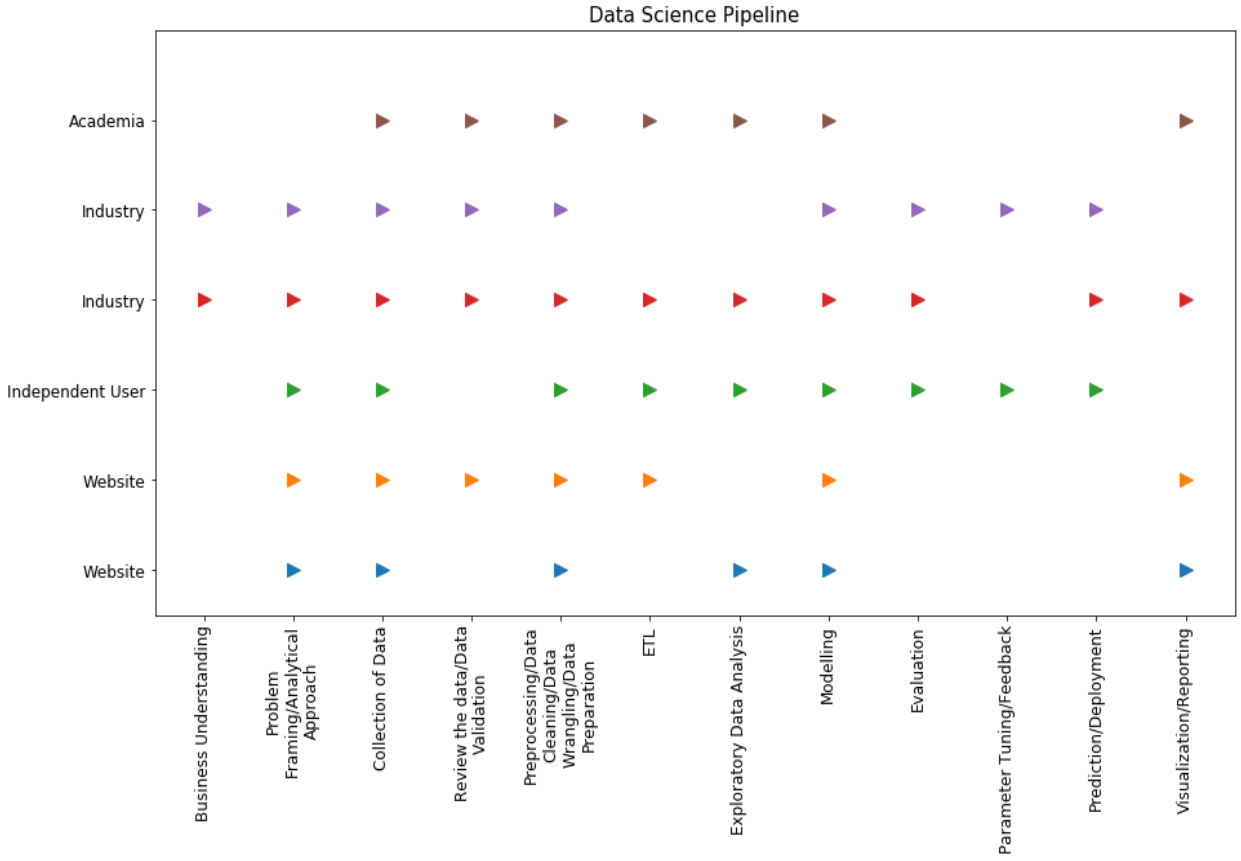

Figure 3.4: Data Science in Theory and Practice

model or interpret the data. While data science pipelines can have more, or less data-related activities depending on the business problem at hand, there is no one prescribed workflow. Figure 3.4 shows the common data science pipelines suggested in both theory (by academics) and practice (by freelance data scientists and organizations).

While business understanding is a part of the data science pipeline in industries, problem understanding/framing is an integral part of every data science pipeline except academia. After the collection of data necessary for the problem, review/validation of data is considered as a separate part in only some of the pipelines which are also the case with Extract, Transform and Load (ETL) step. Data preprocessing is a part of every data science pipeline suggested and I suspect those pipelines which did not emphasize on review/validation or ETL assumed them as a part of data preprocessing. While Modelling is a part of all the suggested data science pipelines, evaluation, parameter tuning, prediction, and visualization/reporting are only emphasized in some of them. This exposes the overlapping nature of the data science activities and inconsistency in the definition of the activities suggested, practised, and required in a data science

pipeline.

3.2.1 Data Science pipeline in Notebooks

Business Understanding and Problem Framing are not programming tasks[7]. Hence, they will only appear at most as markdowns in well-documented notebooks. It is also important to note that while ETL is an important part of the data science pipelines, most of the times, data is already collected and to some extent processed and put together by ETL (Extract, Transform, Load) tools before being loaded into notebooks. In the case of notebooks, it is also possible ETL and data analysis are performed in different notebooks. So, I expect the data science pipelines in a single notebook to contain less data preparation than what would be required in reality. Although reporting form an integral part of the data science pipeline, I do not expect to have reporting as a data science activity in all notebooks except the well-documented ones [Rule et al., 2018]. Similarly, I do not expect deployment as an activity in notebooks, as it is not typically implemented using notebooks.

[7] *https://data.sngular.com/en/art/48/crisp-dm-phase-i-business-understanding*

4

System Design

This chapter discusses the designs decisions, architecture, and the components of the automatic annotation system in detail. In this chapter, I first discuss what is a basic unit of annotation i.e., what constitutes a 'part' of the notebook, which are the parts that perform a data science activity and what are the data science activities that will be used as classification labels. Next, I introduce the system and its components. Finally, I discuss each component and its functionalities in detail.

All the classes, methods and notebooks implemented for the system are discussed in brief in A.1. More details and comments are available in the implemented classes and notebooks.

4.1 System and its Components

4.1.1 Design Decisions

The first step in classification of the parts of the notebook is to identify what a 'part' (granularity) of the notebook is and then generate a feature vector for each part to be classified. In case of a single jupyter notebook, granularity levels could be viewed as cell-based or line-based. The intuitive way of viewing the granularity in a Jupyter notebook is cells. Some users prefer to write a complete functional code block in one cell, while some, prefer a few lines or a single line in one cell. For example, writing import statements in two cells depending on their functionality. In this thesis, cell granularity is chosen for two important reasons. One, it is the actual unit of a jupyter notebook. Two, any data science activity is usually accomplished with few lines of code rather than a single line of code which means annotating a block of code is enough.

As stated in Section 3.1.2, there are different types of cells available in a notebook. As only cells of type *code* actually perform a data science activity, only *code* cells will be annotated in the notebook. *Markdown* or *RawNB* provide the context to the data science activity while *Output* provides the results of a data science activity.

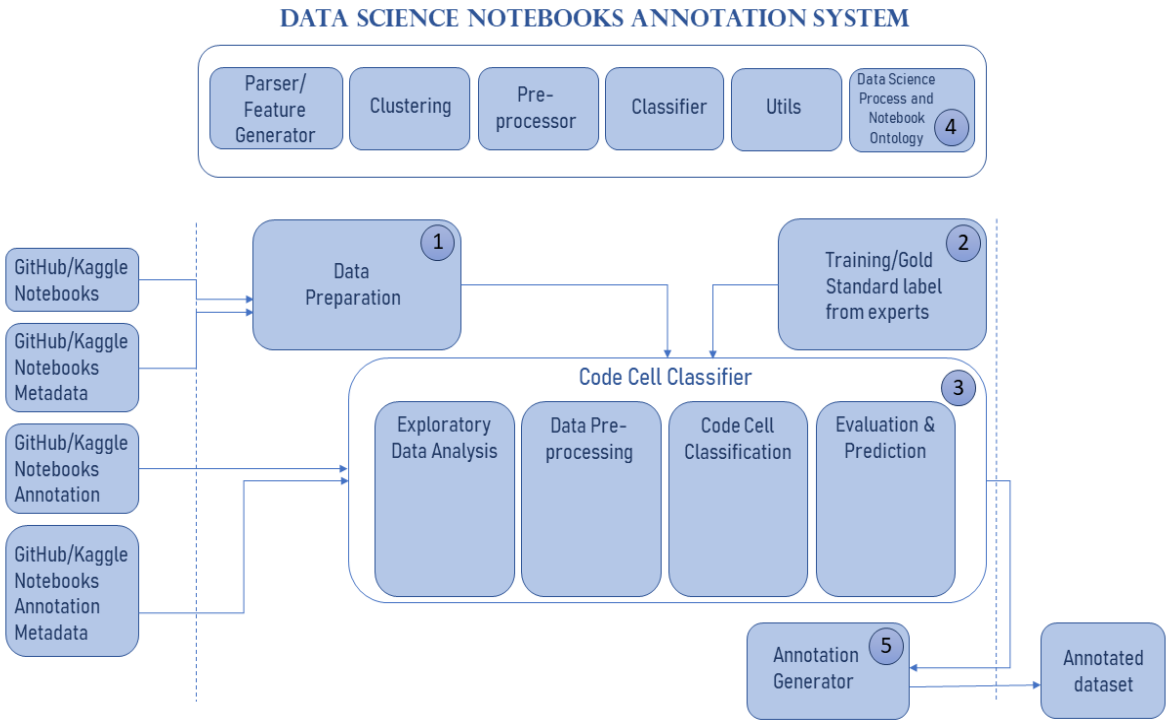

Figure 4.1: Automatic Annotation System for Data Science notebooks

Another important task in classification is to identify the set of classification labels, here, activities in a typical data science pipeline. As I discussed in Section 3.2, there is no one defined approach to what is a standard data science pipeline. Figure 3.4 shows that a pipeline can contain 5-10 activities according to the user: organizations, non-expert users or individual experts. In order to arrive at an appropriate number of steps of a data science pipeline, our own practical knowledge and knowledge from Figure 3.4 is used.

4.1.2 Components

Based on the design decisions, a system for Automatic Annotation of Data Science Notebooks has been developed. In order to provide a clearer picture of the system, I present four important steps as illustrated in Figure 4.1 which depicts the high-level view of the Automatic Annotation System for Data Science Notebooks. Data preparation component prepares the data by extracting features from notebooks, which along with the label from experts is used to train the classifier for code cell classification task. The trained classifier is used to predict the labels for the test set. The output of the classification task for the test data will be then serialized in RDF (see Section 5.6) based on Data Science Process and Notebook Ontology (see Section 4.1.2) and linked data concepts. Each of the components in the system is explained in detail in the following sections.

Data preparation

Data preparation is the first and foremost process in a classification task. In order to perform classification, we need to first extract features from the data science notebooks that will include notebook content, metadata about the notebooks and any other relevant data. Each cell in a notebook thus represents a data point as I consider cell granularity (discussed in Section 4.1.1). Using a feature vector per cell increases the data points in our dataset and I also retain the sequential nature of the cells in a notebook using several other features like execution_count, cell_number etc.

A parser for the notebook (see Section A.1.1) along with various other classes has been implemented to extract the features from each notebook, its metadata and other relevant sources (see A.1 for implementation details). Together they form the Parser/Feature Generator in the system. Feature generation is primarily focused towards notebooks stored in GitHub or Kaggle but are applicable to any notebook of .ipynb type. The features are explained in Section 4.1.2. The data sources used for feature generation are:

1. Jupyter notebooks
2. Metadata information for Jupyter notebooks

Feature Engineering In this section, I explain the feature vector generated using the Parser/Feature Generator (see Section A.1.1) for each cell in a notebook. Each feature vector of a cell apart from the main content of the cell also includes few other features to capture notebook context (indicated by †), i.e., they are common for all the cells for a given notebook (e.g. filename or kernel language). The features are common and have the same definition for all the notebooks created using different workflow systems (Jupyter or Kaggle Kernel) unless otherwise mentioned. The features are divided into four categories based on what they represent. They are Notebook document features, Style features, Statistical features, and Popularity features. In total, there are 60 features which can be either text-based or numeric-based and are explained in detail below.

Notebook document features Notebook document features are features based on the content available in a notebook document.

filename† Filename indicates the name of the notebook file in the dataset.

cell_number Cell number indicates the position of the cell in a given notebook. Cell number along with filename uniquely identifies a cell in the dataset.

execution_count Execution count indicates the order in which a cell was executed by the user in a notebook.

linesofcode Lines of code indicates the total number of lines in a given cell of cell_type code.

linesofcomment Lines of comment indicates the total number of comment lines in a given cell of cell_type code. Any line that starts with '#' is considered as a comment line.

linesofmarkdown Lines of markdown indicates the total number of lines in a given cell of cell_type markdown.

function_count Function count indicates the number of functions in a given cell of cell_type code. Any line that starts with the keyword 'def' is considered to be a function.

variable_count Variable count indicates the number of variables in a given cell of cell_type code. A variable is considered to be any keyword composed of alphanumerics and _ with a '=' to the right. Parameters are not considered.

cell_type Cell type indicates the type of a cell which can be any of the values in [markdown, code, output, raw].

text Text contains either the code content or markdown/raw content depending on the cell_type.

import_text Import text are content of library import statements in a given cell of cell_type code. Any line that starts with 'import..as..' or 'from..import..' or 'from..import..as..' is considered (*python*). Similarly, library imports for other languages are handled according to their respective structure.

comment All the comments in a given code cell. This feature is valid only for code cells.

output_name This indicates the name of the output. This feature is valid for the cells of cell_type output.

output_text This indicates the content of the output. For example, image/png or text/plain. This feature is valid for the cells of cell_type output.

output_type This indicates the type of the output. For example, display_data. This feature is valid for the cells of cell_type output.

code_line_before code_line_before indicates the last line of code in a code cell preceding the current cell. In case of the cells of cell_type markdown or raw_nb, this information is obtained from the code cell preceding it (may or may not be immediately preceding).

code_line_after code_line_after indicates the first line of code in a code cell following the current cell. In case of the cells of cell_type markdown or raw_nb, this information is obtained from the code cell following the current cell (may or may not be immediately succeeding).

markdown_heading markdown_heading indicates the first line of a markdown cell. In case of the cells of cell_type code, this information is obtained from the markdown cell preceding it (may or may not be immediately preceding).

kernel_language† This indicates the kernel language of a given notebook. It is obtained from *['kernelspec']['name']* of notebook metadata (JSON).

language† This indicates the language of the notebook. It is obtained from *['language']* of notebook metadata (JSON).

language_version† language_version indicates the version of the language of a given notebook. It is obtained from *['language_info']['version']* of notebook metadata (JSON).

Style/User features User-based features or Style-based features are features that identify the user and repository information associated with a notebook and is generated from corpus metadata.

repo_id† repo_id is the repository id of the notebook. Each repo_id is unique and belongs to the repository. Each repository may contain one or many notebooks i.e., one or many notebooks may have the same repo_id.

owner† owner indicates the owner id/name of a notebook or the repository containing the notebook.

readme† readme contains the readme information of a repository (in case of GitHub).

Statistical/Metric features Statistical or Metric-based features are features that contain code metrics for a given notebook. I have two set of metrics: standard code metrics implemented through radon[1] and custom metrics generated using classes implemented in Feature Generator for the notebooks.

radon. I generate standard code metrics using radon library[2]. The metrics take into account the whole of the notebook. Hence, the metrics are same for all the cells in a notebook. I convert the .ipynb notebook to .py script and generate the metrics using radon API[3].

[1] *https://pypi.org/project/radon/*
[2] Radon introduction to Code Metrics *https://radon.readthedocs.io/en/latest/intro.html*
[3] *https://radon.readthedocs.io/en/latest/api.html*

r_loc† "The total number of lines of code. It does not necessarily correspond to the number of lines in the file."

r_sloc† "The number of source lines of code - not necessarily corresponding to the LLOC."

r_comments† "The number of comment lines. Multi-line strings are not counted as comment since, to the Python interpreter, they are just strings."

r_multi† "The number of lines which represent multi-line strings."

r_blank_lines† "The number of blank lines (or whitespace-only ones)."

r_single_comments† The number of single line comments.

r_distinct_operators† The number of distinct operators (η_1)

r_distinct_operands† The number of distinct operands (η_2)

r_total_operators† The total number of operators (N_1)

r_total_operands† The total number of operands (N_2)

r_program_vocabulary† $(\eta = \eta\_1 + \eta\_2)$

r_program_length† $(N = N\_1 + N\_2)$

r_calculated_length† $(\widehat{N} = \eta\_1 \log\_2 \eta\_1 + \eta\_2 \log\_2 \eta\_2)$

r_difficulty† $(D = \frac{\eta\_1}{2} \cdot \frac{N\_2}{\eta\_2})$

r_effort† $(E = D \cdot V)$

r_time† $(T = \frac{E}{18})$ seconds (time to program)

r_bugs† $(B = \frac{V}{3000})$ (no of delivered bugs)

r_halstead_volume† $(V = N \log\_2 \eta)$

r_lloc† "The number of logical lines of code. Every logical line of code contains exactly one statement."

r_percentage_of_lines_of_comment† The percent of lines of comment (multiline strings as comment lines are not counted)

r_maintainability_index† Maintainability Index (MI) score in the range 0-100.

r_maintainability_rank† Maintainability rank based on MI score.

1. MI score 100 - 20, Rank A, Maintainability Very high
2. MI score 19 - 10, Rank B, Maintainability Medium
3. MI score 9 - 0, Rank C, Maintainability Extremely low

r_cyclomatic_complexity† Cyclomatic Complexity (CC) score in the range 0-100.

r_cyclomatic_complexity_rank† Cyclomatic Complexity rank based on CC score.

1. CC score 1 - 5, Rank A, Risk low - simple block
2. CC score 6 - 10, Rank B, Risk low - well structured and stable block
3. CC score 11 - 20, Rank C, Risk moderate - slightly complex block
4. CC score 21 - 30, Rank D, Risk moderate - more than moderate - more complex block
5. CC score 31 - 40, Rank E, Risk moderate - high - complex block, alarming
6. CC score 41+, Rank F, Risk moderate - very high - error-prone, unstable block

Custom Metrics. Below custom metrics are generated using class:NotebookMetrics and class:CodeCellMetrics.

tot_loc_per_nb† Total number of lines of code (in cells of cell_type code) in a given notebook.

tot_locomment_per_nb† Total number of lines of comment (in cells of cell_type code) in a given notebook.

tot_function_count_per_nb† Total number of function blocks (in cells of cell_type code) in a given notebook.

tot_variable_count_per_nb† Total number of variables (in cells of cell_type code) in a given notebook.

no_of_code_cells_per_nb† Total number of code cells in a given notebook.

code_tokens_per_nb† Total number of code tokens (tokenized using nltk's word_tokenize) in a given notebook.

no_of_markdown_cells_per_nb† Total number of markdown cells in a given notebook.

markdown_tokens_per_nb† Total number of mardown/raw tokens (tokenized using nltk's word_tokenize) in a given notebook.

Popularity features Popularity-based features are features that indicate the popularity of a notebook. In the case of GitHub, it is indicated using the popularity of the repository containing it. In the case of a Kaggle Kernel, it comes from the fork and upvotes information of the kernel.

For GitHub[4], the popularity features are:

fork_count† Number of times a repository containing the notebook is forked.

star_count† Number of GitHub users who have bookmarked the repository. Stars also indicate the number of appreciation a repository has received.

watcher_count† Number of GitHub users watching the repository to receive notifications on new pull requests and issues that are created for a repository.

For Kaggle[5], the popularity features are:

fork_count† Number of times a Kaggle kernel is forked.

star_count† Number of times a Kaggle kernel is upvoted.

watcher_count† Cells from Kaggle Notebook kernel will have value 0 for the watcher_count feature by default (since there is no such feature in Kaggle).

[4] *https://help.github.com/categories/exploring-projects-on-github/*
[5] *https://www.kaggle.com/docs/kernels#collaborating-on-kernels*

External features

packages_info This feature indicates the man-page information about the libraries imported in a given cell. The information contains the description of libraries as given in an external source: The Python Package Index (pypi)[6].

Annotated dataset preparation for Supervised Learning

Supervised classification requires expert labels which are collected using an annotation experiment to train the model. The following section explains the classification labels applicable to the classification task (classifying cells of a notebook).

Classification Labels Using the analysis of data science activities in Section 3.2, 10 labels were chosen (7 data science activities (indicated by *) + 3 generic activities (indicated by **)). They are explained below:

load_data** Load data is the process of loading a dataset into a jupyter notebook environment. The dataset can be of any type (e.g. .csv, .pkl, .jpg, .png, .hdf5), which once loaded is intended to be used for data analysis or any data science activity.

helper_functions** Helper functions are *import statements* or *other piece of code which are not directly related to the data science activity at hand and rather are useful functions in scripting.* For example, built-in jupyter notebook magic commands. A concrete example would be %matplotlib inline which sets the inline backend so that the output plots are displayed directly below the code cell that executes it. Some more examples would be import pandas as pd, from IPython.display import Audio, %pprint.

comment_only** Comment only labels are applicable to *code* cells which contain only comments. This is not valid for markdown or raw type cells.

data_preprocessing* Data preprocessing includes tasks such as cleaning, instance selection, normalization, transformation, feature extraction, feature selection, etc. Data preprocessing is used to transform the raw data into clean data. It ensures that the data does not contain irrelevant, redundant and inconsistent data. The product of data preprocessing is the final training set.[7]

data_exploration* Data exploration[8] or Exploratory Data Analysis is an approach to initial data analysis whereby a data scientist or an analyst uses techniques to inspect what is in a dataset and understand the nature and characteristics of the data. Summarization

[6] *https://pypi.org/*
[7] *https://en.wikipedia.org/wiki/Data_pre-processing*
[8] *https://en.wikipedia.org/wiki/Data_exploration*

of data's main characteristics and clearly emerging patterns are helpful in deciding the models and hypotheses. Data exploration often involves visual exploration.

modelling* Modelling is the process of applying (/fitting) various learning or statistical models and algorithms on the data in order to 'perform a specific task without explicit restrictions, relying on models and inferences instead'[9]. Modelling can be descriptive, predictive or prescriptive depending on the business problem.

prediction* Prediction is an important step as many of the Data science tasks are predictive modelling tasks. Once the model has been trained, it can be used to predict the output on a new set of data.

evaluation* Evaluation is the process of evaluating a model using various evaluation metrics like the goodness of fit between model and data, accuracy, f1score and so on. Evaluation is also done to compare different models, in the context of model selection, and to evaluate how accurate are the predictions (associated with a specific model and data set)[10].

result_visualization* Result visualization is the data visualization step of a Data Science pipeline. It is the graphical representation of information and data. By using visual elements like charts, graphs, it enables decision-makers to see the analytics, model performance, results visually and understand trends, detect outliers and patterns in data[11].

save_results* Save results is the process of serializing and storing the results from a data science activity.

Simulation A simulation of expert annotation was done with a mixed group of experts (Engineering, Finance, Computer Science) who have knowledge about data science. The observations from the simulation regarding the availability of instructions, clarity of tasks assigned, availability of resources, and other general comments were taken into consideration while designing the expert annotation task. The purpose of the simulation was to understand and identify the clarity of instructions, completeness of classification labels, output format, and the time required to complete annotation.

Annotation Experiment: Training labels by experts Code cell classification task requires labelled dataset. Data science activities are the class labels5.4.1 and are essential in the supervised learning[12] task of a classifier. As the thesis investigates both Single-label Multiclass and Multilabel Multiclass classification, annotation task is designed in

[9] *https://en.wikipedia.org/wiki/Machine_learning#Models*
[10] *https://www.oreilly.com/data/free/evaluating-machine-learning-models.csp* for more information
[11] *https://www.tableau.com/learn/articles/data-visualization*
[12] *https://en.wikipedia.org/wiki/Supervised_learning*

such a way that all the relevant labels and a primary_label for a given classification instance is captured. To put it concretely, each code cell will be assigned a set of labels: [primary_label, relevant_label1, relevant_label2...relevant_labeln]. Figure 4.2 shows a sample annotation template with classification labels recorded. primary_label will be used in Singlelabel Multiclass classification and the complete set of labels will be used in Multilabel Multiclass classification.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	filename	cell_number	cell_type	starting_text	primary_label	load_data	helper_	data_prepr	data_explc	modelling	prediction	evaluation	result_visu	save_results	comment_	other_labe	notes
2	cancer-tre	0	markdown	Hello all, im a nebie here and i'll try to explain you what i'v		0	0	0	0	0	0	0	0	0	0	0	
3	cancer-tre	1	code	import numpy as np # linear algebra	helper_functions	0	0	0	0	0	0	0	0	0	0	0	
4	cancer-tre	2	code	import numpy as np # linear algebra	helper_functions	0	0	0	0	0	0	0	0	0	0	0	
5	cancer-tre	3	code	train_variants_df = pd.read_csv("../input/	load_data	0	0	0	0	0	0	0	0	0	0	0	
6	cancer-tre	4	code	train_text_df.shape,test_text_df.shape	data_exploration	0	0	0	0	0	0	0	0	0	0	0	
7	cancer-tre	5	code	train_variants_df.shape,test_variants_df.	data_exploration	0	0	0	0	0	0	0	0	0	0	0	
8	cancer-tre	6	code	train_variants_df.head(3)	data_exploration	0	0	0	0	0	0	0	0	0	0	0	
9	cancer-tre	7	code	train_text_df.head(3)	data_exploration	0	0	0	0	0	0	0	0	0	0	0	
10	cancer-tre	8	code	gene_group = train_variants_df.groupby(	data_preprocessin	0	0	0	0	0	0	0	0	0	0	0	
11	cancer-tre	9	code	test_variants_df.head(3)	data_exploration	0	0	0	0	0	0	0	0	0	0	0	
12	cancer-tre	10	code	test_text_df.head(3)	data_exploration	0	0	0	0	0	0	0	0	0	0	0	
13	cancer-tre	11	code	train_text_df.Text[0]	data_exploration	0	0	0	0	0	0	0	0	0	0	0	
14	cancer-tre	12	code	train_variants_df.Class.unique()	data_preprocessin	0	0	0	0	0	0	0	0	0	0	0	
15	cancer-tre	13	code	plt.figure(figsize=(15,5))	data_exploration	0	0	0	0	0	0	0	0	0	0	0	
16	cancer-tre	14	code	print(len(train_variants_df.Gene.unique(	data_exploration	0	0	0	0	0	0	0	0	0	0	0	
17	cancer-tre	15	code	train_df = pd.merge(train_text_df,train_v	data_preprocessin	0	0	0	0	0	0	0	0	0	0	0	

Figure 4.2: A sample annotation template with classification labels recorded.

The following annotation task is set up in order to obtain the class labels for a data science notebook based on a given set of guidelines and questions. The class labels are to be obtained from data science experts manually inspecting and annotating the data science notebook.

Goal. The goal is to annotate a given data science notebook i.e, assign one or more classification labels (data science activities) to each code cell based on the guidelinesA.3.

Annotator Profile. The annotator is a data science expert with considerable experience in the activities involved in a Data Science project. The annotator can also be a computer scientist/statistician or someone who is involved in a computational discipline and has some practical experience with data science projects.

Class labels. Load Data (load_data), Helper Functions (helper_functions), Comment Only (comment_only), Data Preprocessing (data_preprocessing), Data Exploration (data_exploration), Modelling (modelling), Evaluation (evaluation), Prediction (prediction), Result Visualization (result_visualization), Save Results (save_results).

Output. Annotators were asked to indicate their labels for each *code* cell in a notebook in an annotation template <filename>.xlsm or <filename>.csv file (see Figure for the format), for each notebook separately. In addition, the experts were asked to indicate how confident (in the scale of 0-100) they are about the annotation and how well-written (understandable, clarity of process, clarity of code) is the notebook (in the scale of 0-10).

Code Cell Classifier

The classification task of our annotation system, code cell classification (refer to 5.5) is implemented by instantiating various state of the art classifiers available in scikit-learn. The classifiers are discussed in brief in Section A.5. Code cell classification aims to implement and evaluate the classifier that best classifies the code cells in data science notebooks according to the data science activity. I explore both Singlelabel and Multilabel classification methods while using binary relevance (OnevsRest in scikit-learn) as the classification technique for Multilabel classification. Figure 4.3 illustrates a simple notebook annotated according to the data science activities.

Data Science Process and Notebook Ontology

"The W3C Web Ontology Language (OWL) is a Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things."[13]. Ontologies (or vocabularies) are documents that define and describe the concepts and relationships of a particular domain and exist as OWL/RDF etc. There are many ontologies available[14] that can be reused to represent the knowledge in

[13] *https://www.w3.org/2001/sw/wiki/OWL*
[14] List of Ontologies *https://www.w3.org/wiki/Lists_of_ontologies*

Figure 4.3: A simple notebook and its annotations.

the world wide web[15].

Data science activities representation in notebooks requires a vocabulary that define concepts about notebooks, data science activities and the relationship between them. A new ontology, Data Science Process and Notebook Ontology (***no***) (see Section A.2) has been developed that defines the concepts of notebooks and data science activities I have identified in a data science pipeline. I also reuse RDF ontology and OWL ontologies wherever possible.

[15]Refer *https://www.w3.org/standards/semanticweb/ontology* for more information.

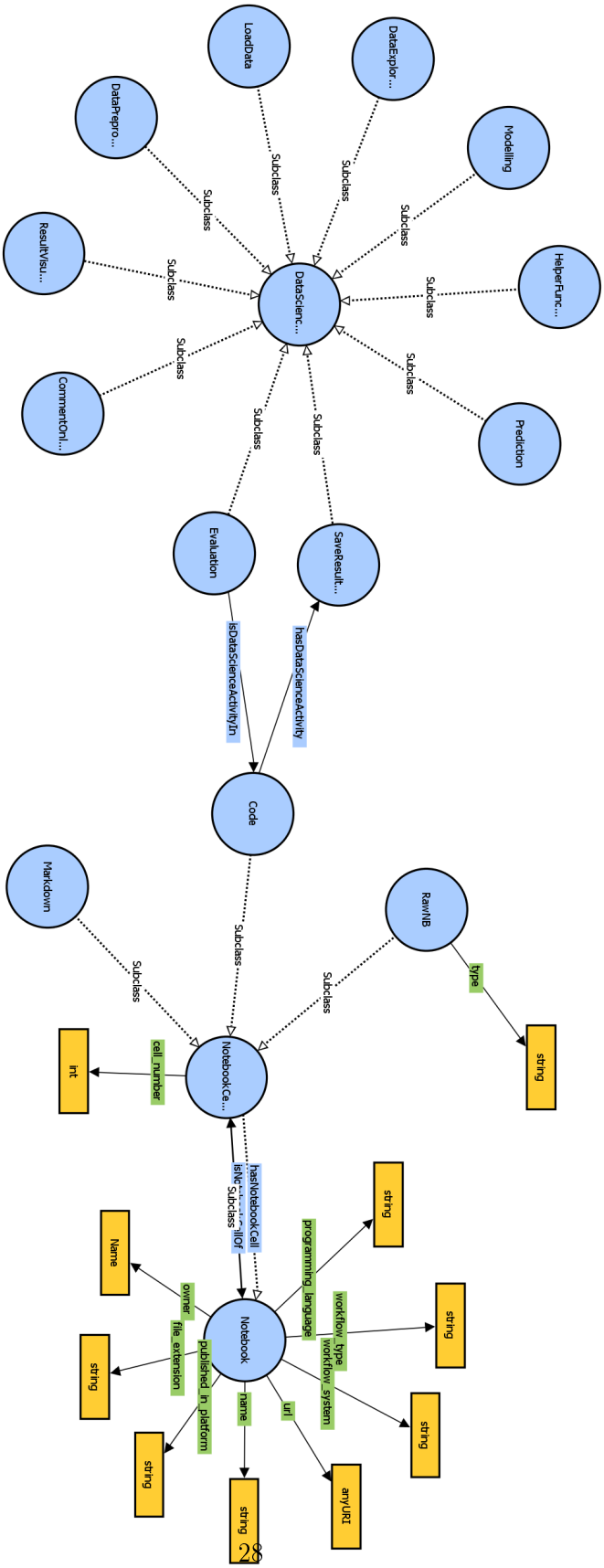

Figure 4.4: Data Science Process and Notebook Ontology

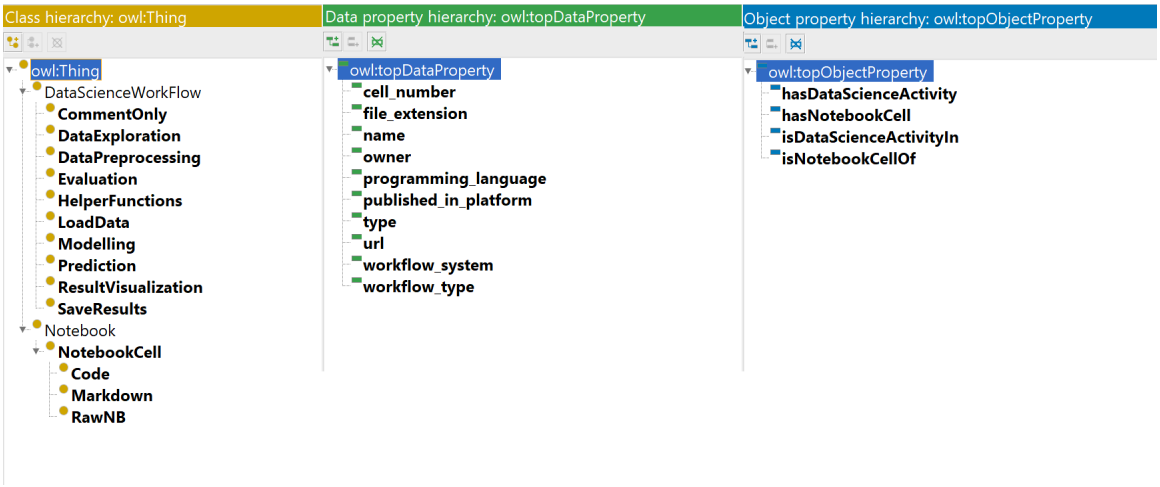

Figure 4.5: Data Science Process and Notebook Ontology: Classes and Properties

The ***no*** ontology developed contains concepts, data properties and object properties relevant to semantically annotate data science notebooks and is described below. The Ontology is designed using Protégé[16] and is represented in .owl format.

Figure 4.4 shows the ontology ***no*** visualized using ProtégéVOWL[17].

Concepts and Relationships

Figure 4.5 shows concepts (classes and subclasses) and relationships (data properties and object properties) of the ***no*** ontology as in Protégé.

Concepts

1. Concepts for Notebook: Notebook, NotebookCell, Markdown, RawNB, Code .
2. Concepts for Data science process: DataScienceWorkflow, HelperFunctions, LoadData, DataPreprocessing, DataExploration, Modelling, Evaluation, Prediction, ResultVisualization, SaveResults.

Data Properties cell_number, file_extension, name, owner, programming_language, published_in_platform, type, url, workflow_system, workflow_type

Object Properties hasDataScienceActivity (n:n), hasNotebookCell (1:n), isDataScienceActivityIn (n:n), isNotebookCellOf (n:1)

[16] *https://protege.stanford.edu/*
[17] *http://vowl.visualdataweb.org/protegevowl.html*

Annotation Generator

A notebook (refer to Section A.1.7) has been implemented to produce annotated dataset based on the ontology developed given the data science activity labels for a set of data points. Annotation is serialized in the RDF format and follows the linked data concept.

5

Experiments and Results

In this chapter, I present the data, methodology, experimental setup and the results of the empirical evaluation that I accomplished for the automatic annotation task. Moreover, I showcase the RDF annotation process to generate a data set that represents notebooks and its data science labels using Data Science Process and Notebook Ontology.

5.1 Data

In this chapter, I present the corpora of Notebook documents (or "notebooks") containing GitHub Notebook documents corpus and Kaggle Notebook documents corpus. I then discuss in brief the characteristics of the dataset chosen for the experiment.

5.1.1 Corpora of Jupyter Notebook Documents

Jupyter notebook corpora used in this thesis contains notebooks retrieved from two sources: GitHub and Kaggle. All the notebooks have been created using Jupyter workflow system that supports various programming language kernels.

GitHub corpus

GitHub corpus contains a part of the ~1 million Jupyter notebooks[1] dataset created for exploration of how people use narrative text in Jupyter Notebooks. The dataset is published by the Design Lab[2] team at UC San Diego[3] and also includes metadata information of the repository [Rule et al., 2018]. The metadata information of the repository has been used to retrieve owner information, repository information including README for the GitHub notebooks. Other metadata information for GitHub corpus (refer to Section 5.1.1) such as fork, star, and watcher counts have been retrieved using the implemented classes (see Section A.1). Python notebooks account for a large portion of the data (97%) while R and Julia are the other prominent languages (see Figure 5.1) in the

[1] Dataset *https://library.ucsd.edu/dc/collection/bb6931851t*
[2] *https://designlab.ucsd.edu/*
[3] *https://designlab.ucsd.edu/*

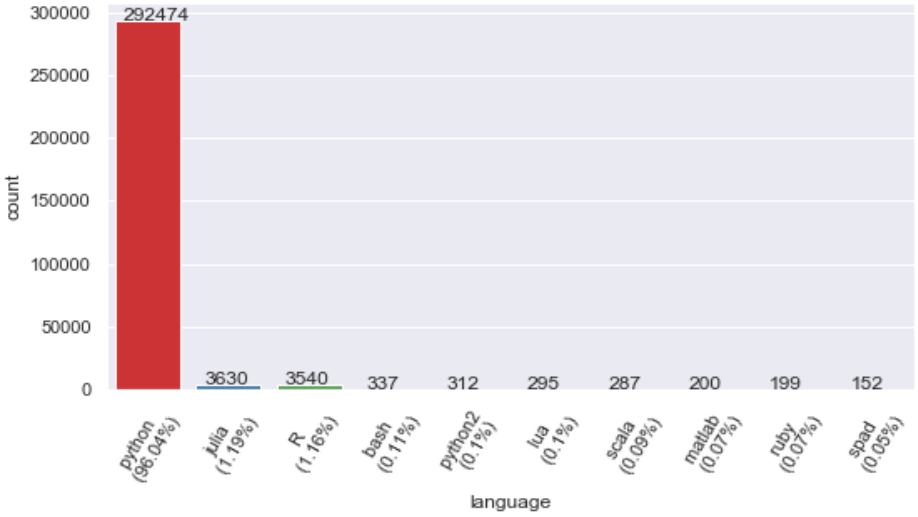

Figure 5.1: Language composition of GitHub Corpus

dataset. The notebooks are obtained from over 1000 repositories which are generated by academics, data science practitioners etc.

Kaggle corpus

Kaggle corpus contains publicly available Kaggle notebooks[4] retrieved for a set of search keywords[5] using Kaggle API. Metadata results for the notebooks including the links to the Kaggle notebooks are retrieved using Kaggle API[6]. *Value:ref* in the metadata is then used to download the notebook. Features like fork and upvoters have been retrieved using classes implemented for the thesis. A total of 111 notebooks are available in the corpus of which 100% are of programming language python.

A Kaggle notebook document is one of the possible types of Kaggle kernel (others being RMarkdown Scripts and Scripts). Kaggle notebook kernel supports Python and R as a programming language and consist of a sequence of cells, where each cell is formatted in either markdown (for writing text) or in a programming language of one's choice (for writing code)[7].

5.1.2 Selection of Dataset

[4] *https://www.kaggle.com/docs/kernels*
[5] More details in the implementation
[6] *https://github.com/Kaggle/kaggle-api*
[7] *https://www.kaggle.com/docs/kernels*

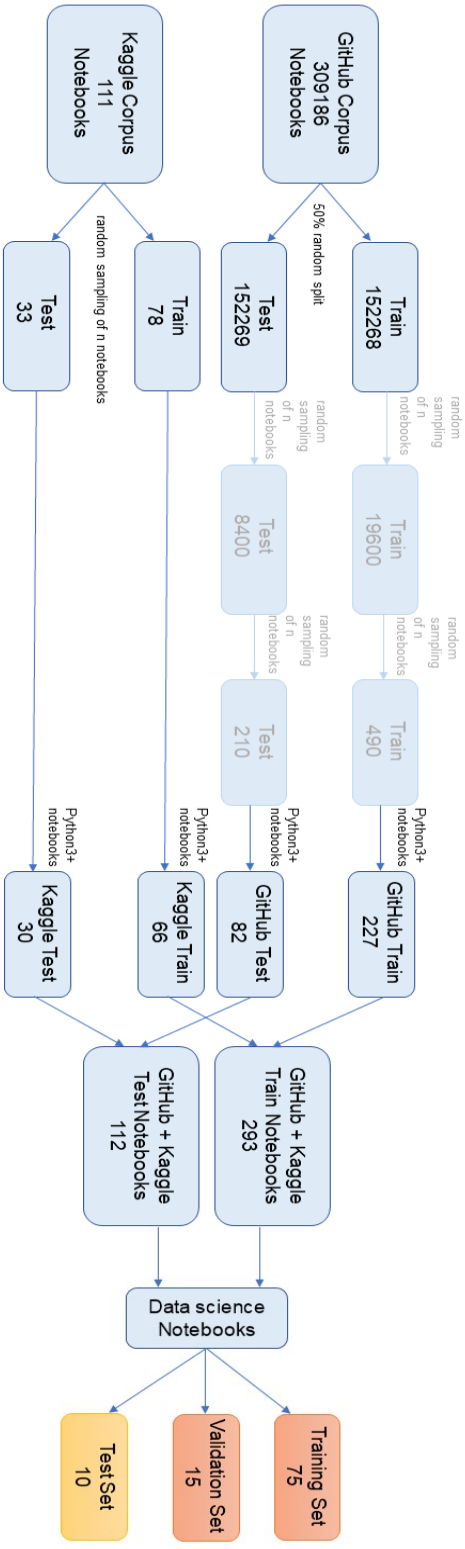

Figure 5.2: Dataset Selection from the Corpus

I chose a total of 100 notebooks containing GitHub and Kaggle notebooks, out of which 90 notebooks form training dataset (out of which 15 form validation dataset) and 10 notebooks form test dataset. Figure 5.2 shows the dataset selection stages in detail.

Characteristics of the dataset

The corpus of both GitHub and Kaggle contains Jupyter notebooks that were employed for different purposes. As the thesis focuses on data science notebooks, a smaller set of notebooks were chosen according to the task the notebooks perform. The dataset chosen from the corpora satisfies following conditions.

Kernel language The corpora contains notebooks composed of various programming languages. Figure 5.3 shows the composition of languages in the dataset, both training, and the test set. All the notebooks selected for the experiment use Python as their kernel language while the parser developed works also on a notebook of any kernel language.

Purpose Only notebooks that were developed for the purpose of a data science task are considered. I identified this manually by inspecting the notebooks.

5.2 Methodology

For the evaluation of the automatic annotation system, I followed the following methodology.

1. I generate features from the GitHub and Kaggle notebooks dataset selected for the experiment (see Section 5.3).

2. I prepare classification labels for the training (including validation) and test dataset. I also get the classification labels for the notebooks from data science experts using the annotation experiment designed (see Section 4.1.2).

3. I perform an exploratory analysis to understand the characteristics of the dataset in various dimensions (see Section 5.4) using statistical and unsupervised machine learning methods.

4. After selecting the data points for classification from the dataset, i.e., data points of cell_type *code*, I split the dataset into training (70% of the notebooks dataset), validation (20% of the notebooks dataset) and test (10% of the notebooks dataset) set.

5. I select the text-based features and perform data preprocessing using the methods available in the Preprocessing class for training, validation, and test set.

6. After which, I vectorize the text features of the dataset using Tfidf method.

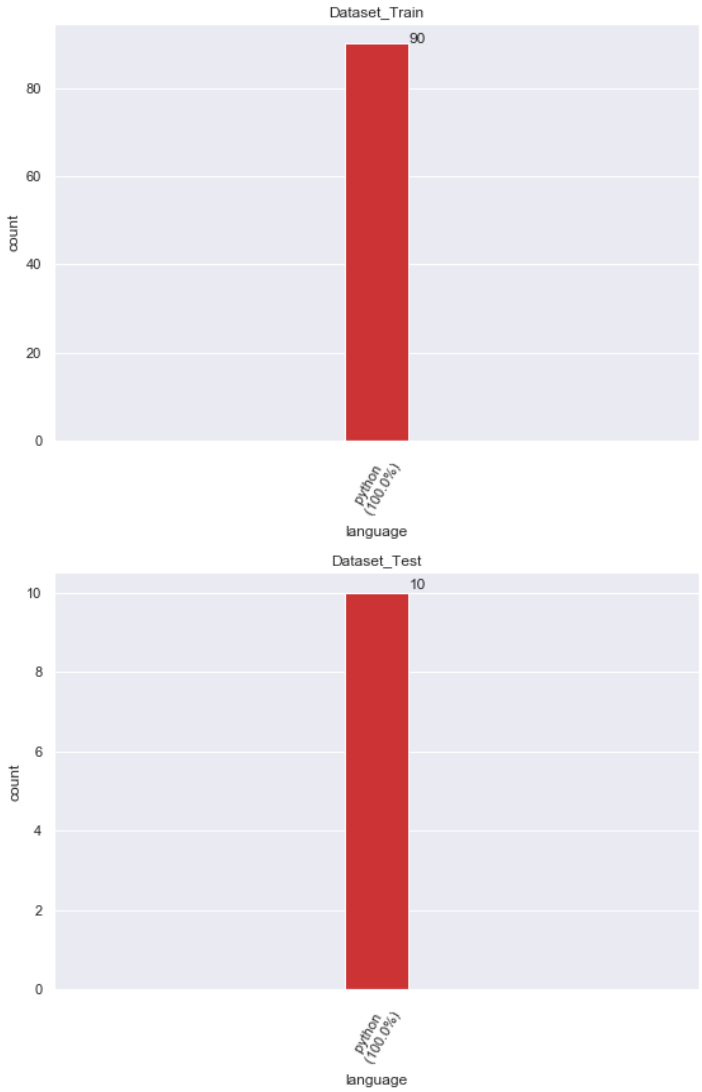

Figure 5.3: Language composition of the Dataset for Experiment

7. I then perform feature selection using chi2[8] method.
8. I select the statistical features as required to complete the feature vector.
9. I perform standardization of the features to represent both text-based and statistical-based features on the same scale.
10. I investigate several classification methods using both Singlelabel and Multilabel Multiclass classification for the thesis and identify the best classifier given the setup (see Section 5.5 for results). (This step is one-time evaluation.)
11. Given the best classifier, I evaluate the estimators (best hyperparameters) using GridSearchCV[9]. GridSearchCV uses PredefinedSplit cross-validator[10].
12. I train the classifier using the training set (classifier parameters are identified by GridSearchCV in the previous step).
13. Using the trained classifier, I predict the classification labels for the test set.
14. At the end of the classification task, I produce a dataset with RDF annotations using predicted labels and notebook information based on the Data Science process and Notebook Ontology.
15. I verify that the dataset with RDF annotation is semantically analysable using SPARQL queries.
16. Based on the observations from the classification task, I answer the research questions.
17. I discuss in detail about feature importance and other relevant points on classification accuracy.

5.3 Data Preparation

This section discusses the data preparation process of notebooks dataset which includes feature generation and preparation of classification labels for the dataset.

5.3.1 Feature Generation

I generated the features explained in Section 4.1.2 for the dataset of 100 data science notebooks using Notebook Parser and classes implemented in Feature Generator (see Section A.1.1). Classes used and their characteristics are given below:

1. CellFeatures: For each cell in the notebook, a feature vector is generated.

[8] *https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.chi2.html*
[9] *https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html*
[10] *https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.PredefinedSplit.html*

2. PypiPackagesInformationFeatures: For each cell in the notebook, a feature vector is generated.

3. StyleFeatures: The feature values generated using this class are common for all the cells of a notebook. For Kaggle, the author is considered as the owner. Kaggle will not have repo_id and readme (not applicable) and will have a *None* value.

4. CodeMetrics Class: The feature values generated using this class are common for all the cells of a notebook.

5. NotebookMetrics: The feature values generated using this class are common for all the cells of a notebook.

6. CodeCellMetrics: The feature values generated using this class are common for all the cells of a notebook.

7. PopularityMetrics: The feature values generated using this class are common for all the cells of a notebook. Since popularity metrics require metadata as in GitHub corpus format, a python function for Kaggle metrics is implemented separately. In Kaggle, totalVotes is the star_count and forks is fork_count. I keep a constant $'0'$ value for watcher_count as Kaggle does not have such a metric.

A Jupyter notebook has been implemented for data preparation task that generates and puts together all the features. It splits the data into training, validation, and test dataset to be used for further analysis/machine learning. Totally the dataset contains 60 features including text-based features + 11 labels (for Singlelabel[11] and Multilabel classification[12]).

It is important to emphasize that, only data points that have cell_type *code* will be taken further for the classification task.

5.3.2 Classification Labels

A total of 100 notebooks were annotated by data science experts based on the annotation experiment designed (explained in Section 4.1.2). I annotated a total of 100 notebooks which is used in the classification task. Two more data science experts annotated a total of 50 notebooks which is used for assessing the quality of annotations, while I reviewed the rest of the 50 notebooks using the knowledge from the annotations by the other two experts. Each *code* cell is assigned a primary_label along with other relevant labels (discussed in Section 4.1.2). Annotation by experts together with the features generated from the notebooks form the dataset for the classification task.

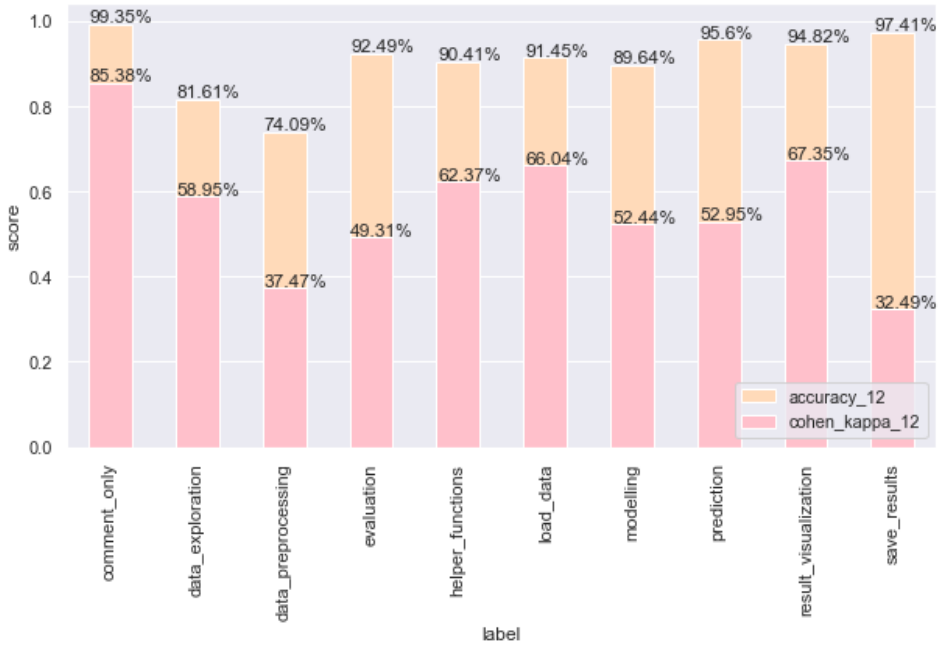

Figure 5.4: Inter annotator agreement scores: Main annotator and Annotator 1

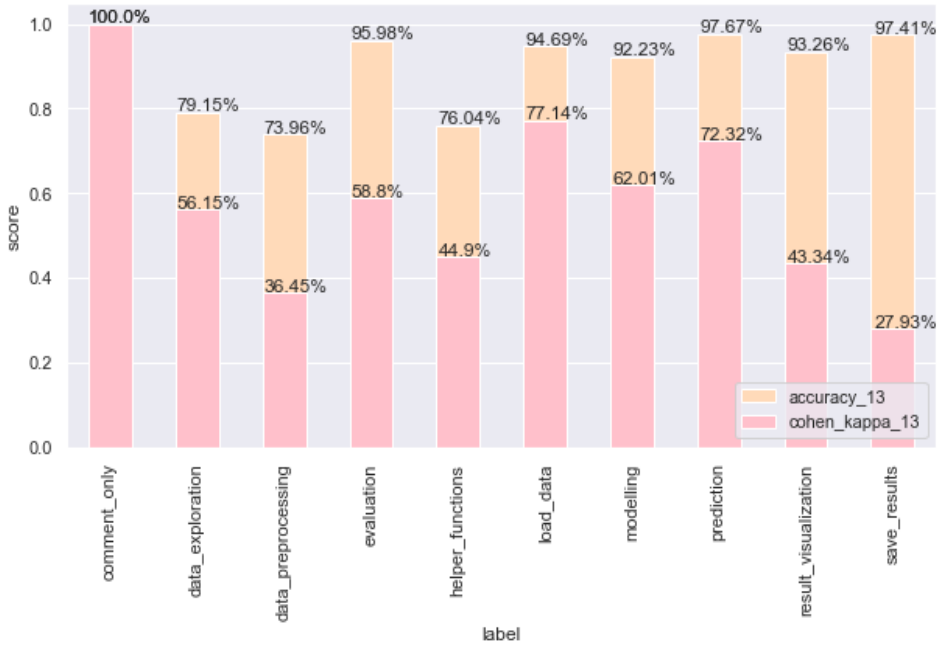

Figure 5.5: Inter annotator agreement scores: Main annotator and Annotator 2

	cohen_kappa	accuracy
comment_only	0.8538	Very good
data_exploration	0.5895	Moderate
data_preprocessing	0.3747	Fair
result_visualization	0.6735	Good
evaluation	0.4931	Moderate
helper_functions	0.6237	Good
load_data	0.6604	Good
modelling	0.5244	Moderate
prediction	0.5295	Moderate
save_results	0.3249	Fair

Table 5.1: Cohen's kappa score for classification labels: Main annotator & Annotator 1

Evaluation of Annotation Quality: Inter-Annotator Agreement

For evaluating the inter-annotator agreement between the annotations, we used Cohen's kappa score from scikit. Cohen's kappa is "a score that expresses the level of agreement between two annotators on a classification problem"[13]. I have also evaluated the accuracy[14] along with the Cohen's kappa score. Both the measures are evaluated for each label in the classification labels set. Figure 5.4 and 5.5 shows the Cohen's kappa score and the accuracy for each label in the main annotation against *annotator1* and *annotator2* respectively. I have a very good accuracy score for most of the labels while Cohen's kappa score perform moderate to above-moderate for most of the labels.

For the kappa score, based on the interpretation of kappa using [Altman, 1990], the agreement between main annotator and *annotator1* for labels are given in Table 5.1. Table 5.2 shows the cohen_kappa score for labels between main annotator and *annotator2*

5.4 Exploratory Analysis of the Dataset (EDA)

In this section, I discuss the exploratory analysis done on the dataset to understand the characteristics of the notebooks, its metadata, labels etc using statistical and unsupervised learning methods. The exploratory analysis is focused towards the following questions:

Statistical Analysis

[11] Singlelabel classification and Singlelabel Multiclass classification are used interchangeably
[12] Multilabel classification and Multilabel Multiclass classification are used interchangeably
[13] *https://scikit-learn.org/stable/modules/generated/sklearn.metrics.cohen_kappa_score.html*
[14] *https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html*

	cohen_kappa	accuracy
comment_only	1.0	Very good
data_exploration	0.5615	Moderate
data_preprocessing	0.3645	Fair
result_visualization	0.4334	Moderate
evaluation	0.588	Moderate
helper_functions	0.449	Moderate
load_data	0.7714	Good
modelling	0.6201	Good
prediction	0.7232	Good
save_results	0.2793	Fair

Table 5.2: Cohen's kappa score for classification labels: Main annotator & Annotator 2

1. What are the general characteristics of notebooks and its cells?
2. What are the general characteristics and composition of cells of cell_type code in the notebooks dataset?
3. What is the composition of external libraries in data science notebooks?
4. What are the general characteristics of the classification labels in the dataset?

Unsupervised Methods

1. What is the number of latent topics in data science notebooks?
   a) Based on code-markdown-raw data
   b) Based on code data
   c) Based on only import statements

5.4.1 Statistical Analysis

Notebook Characterisitics

Exploration of the characterisitics of Jupyter notebook dataset is performed to understand the general characteristics of the dataset. Figure 5.6 shows the number of code cells vs. markdown cells in a notebook and Figure 5.7 shows the total number of tokens in code cells vs. total number of tokens in markdown cells. From both the figures, we see that the markdown cells and markdown content are considerably less in most of the notebooks. This result for data science notebooks is in line with the findings from the paper [Rule et al., 2018] that there is not enough explanation or reasoning of the results in the notebooks. Figure 5.8 shows the cyclomatic complexity composition of the notebooks from low risk-A to very high risk-F.

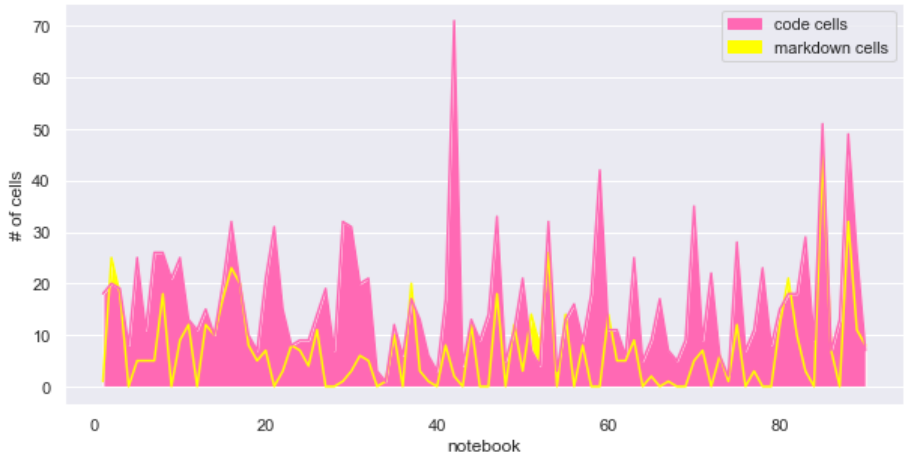

Figure 5.6: No of code cells vs markdown cells per notebook

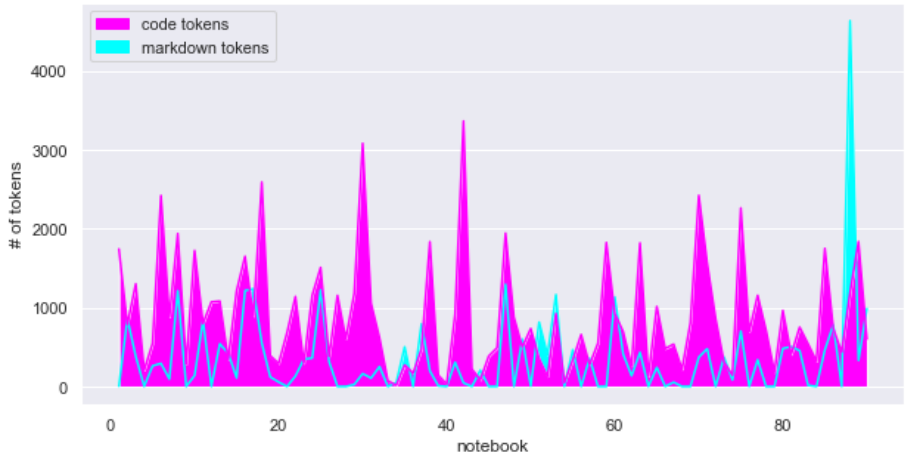

Figure 5.7: No of code tokens vs markdown tokens per notebook

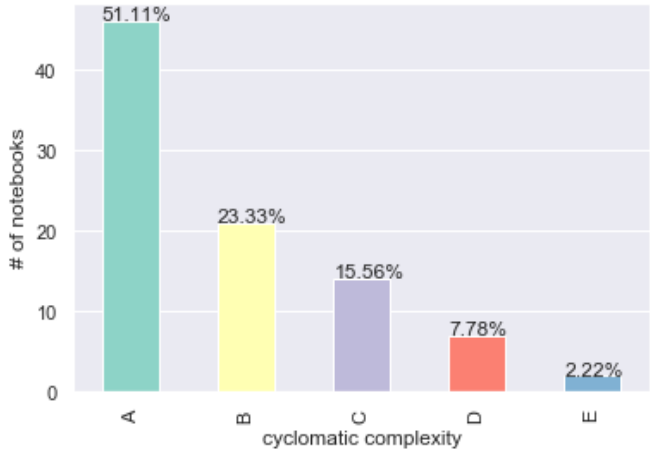

Figure 5.8: Cyclomatic Complexity

Code Cells Characterisitics

To understand the characteristics of code cells in notebooks, I use the datapoints of cell_type *code* from the dataset and $groupby(filename)$ to create an aggregate of various features. Characteristics of the dataset are explored in various dimensions (per notebook): number of code cells, number of lines of comment, number of lines of code, number of variables, and number of functions.

Figure 5.9 shows that most of the notebooks (90%) contain 0 to 31 code cells. It also shows that each notebook contains 68.9% of the times 0-55 variables and around 76.6% of the notebooks have only 0-3 functions in them. The interesting finding is that 85.6% of the notebooks had up to 199 lines of code and around 40% of the notebooks have more than 99 lines of code showing a great variation in the dataset.

External libraries in the notebooks

As a part of EDA, I also analysed the composition of external libraries in the data science notebooks. Figure 5.10 shows that 6.86% of the times a library is imported, it is *numpy*. It is important to note that I have not eliminated the duplicate import statements downloading the same library multiple times. This is helpful in analysing different functions imported from same libraries. From the same figure, we see that 11.45% of the times *scikit-learn* and 5.95% of the times *matplotlib* is imported. The highest percentage of scikit also reveals that multiple scikit functions are downloaded in a single notebook. In total, 30.06% of the libraries imported are *pandas* and the above-mentioned libraries which are as expected as these are considered necessary libraries for a data science task. In Figure 5.11, we observe that 30% of the times, up to 3 libraries are imported in every notebook (ignoring duplicates). This is in line with our previous observation that some libraries are necessary for a data science task.

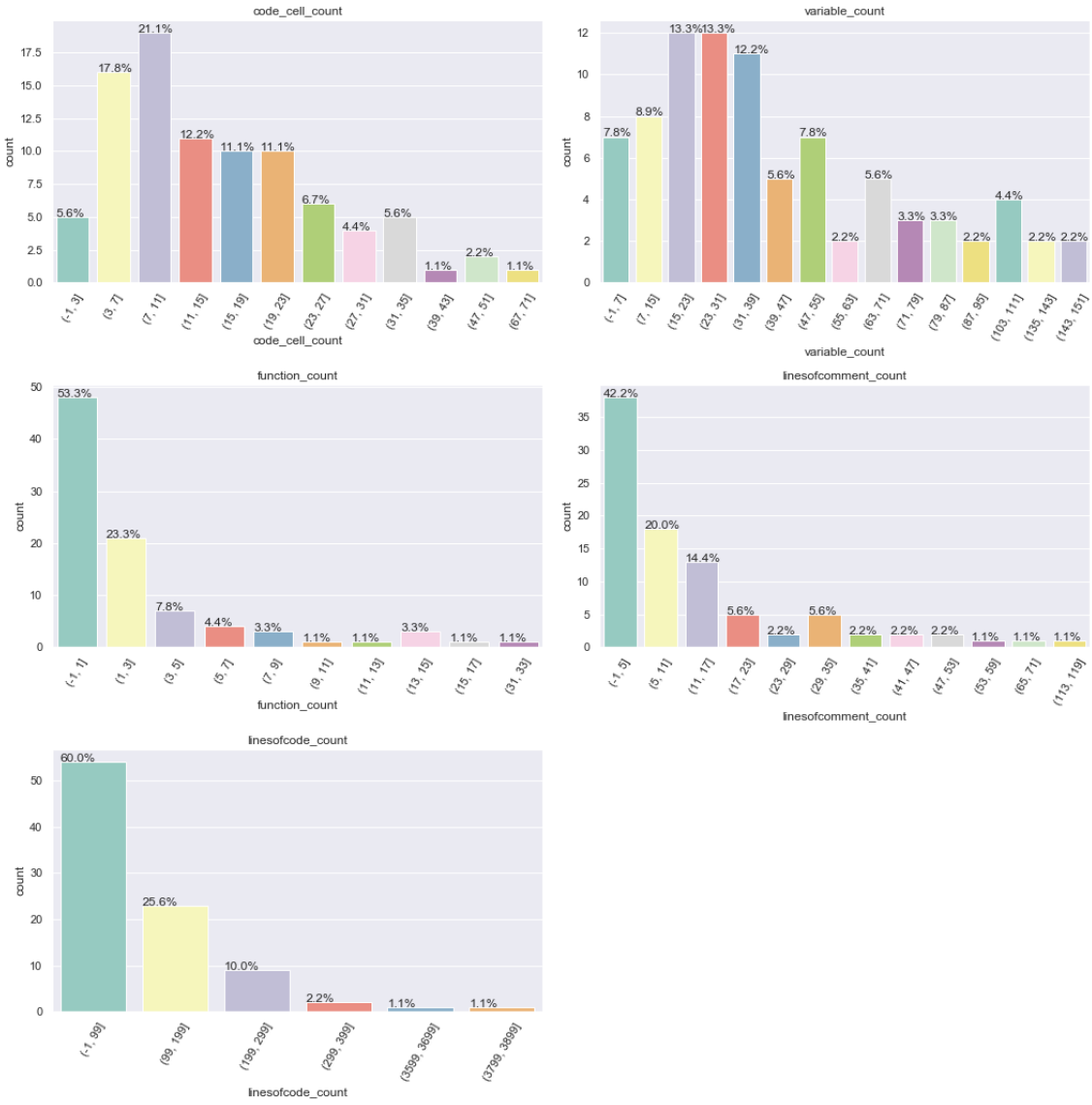

Figure 5.9: Code cell characterisitics per notebook

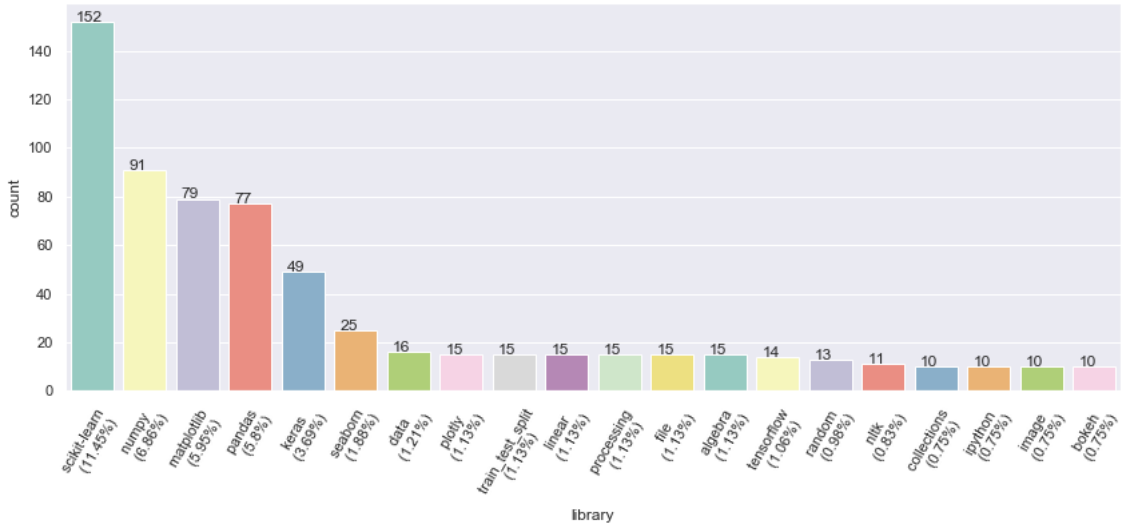

Figure 5.10: Top 20 libraries imported among the notebooks

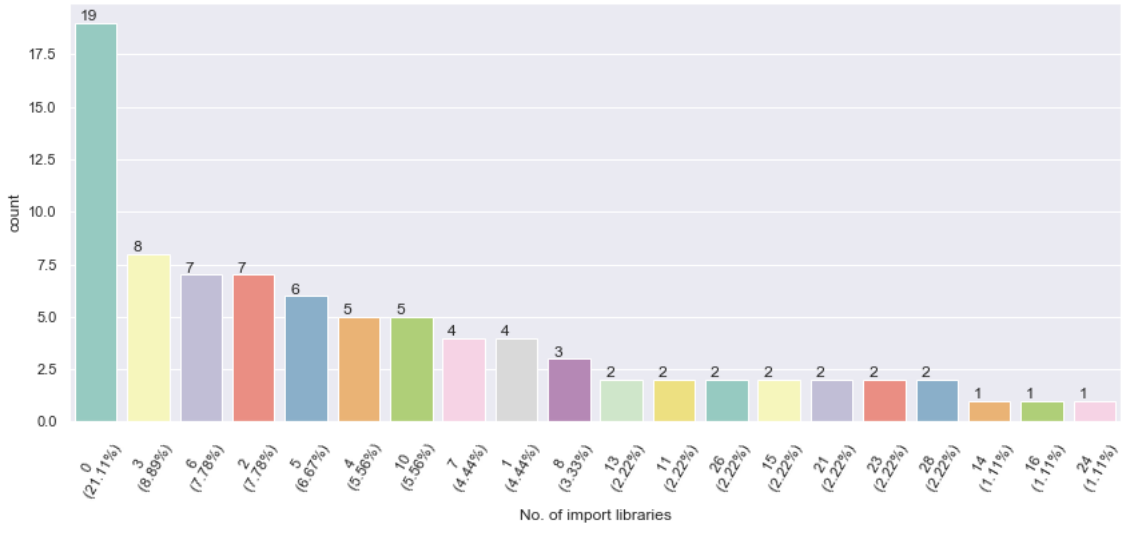

Figure 5.11: Number of libraries imported per notebook

Figure 5.12: Top 1000 words in dataset:code-markdown-rawnb

The exploratory analysis shows patterns with respect to the classification labels and statistical features I have generated and has the potential to improve the accuracy of the classification models. Please note that more analyses are found in the notebooks.

Lexical Universe View

Figure 5.12 shows the top 1000 words occurring in the dataset:code-markdown-rawnb and Figure 5.13 shows the top 1000 words occurring in the data:code.

Classification Labels

In this section, I discuss the exploratory analysis focused towards classification labels. Figure 5.14 shows that around 77% of the cells have only one data science activity while 23% of the cells have more than one labels. From the annotation experiment, I observed that the main reason for having more than one labels assigned to a cell is predominantly due to overlapping and non-linear data science activities. It is important to take note that, a programmer's coding style is also another top reason.

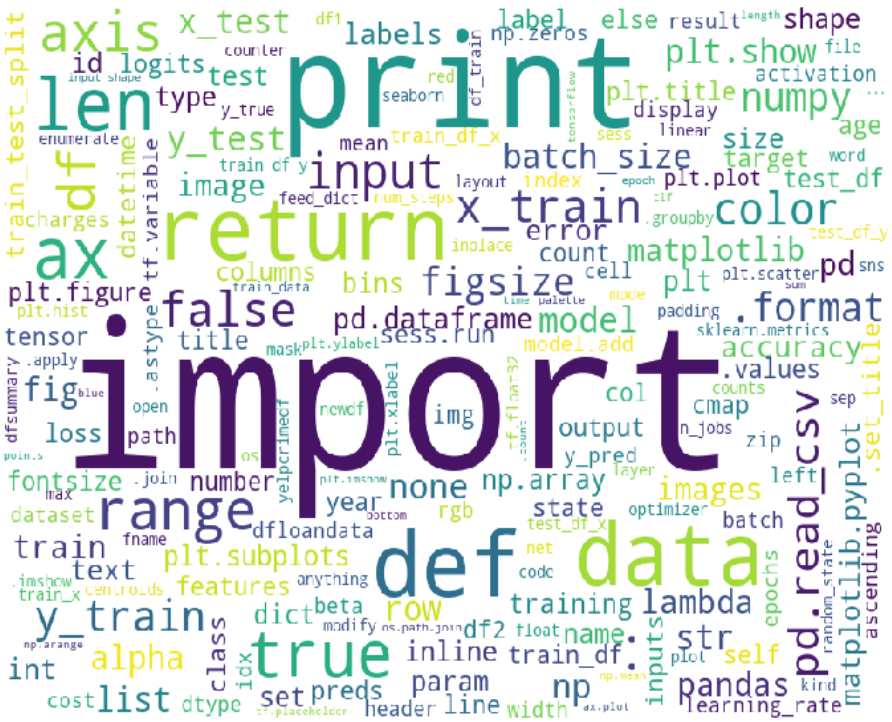

Figure 5.13: Top 1000 words in dataset:code

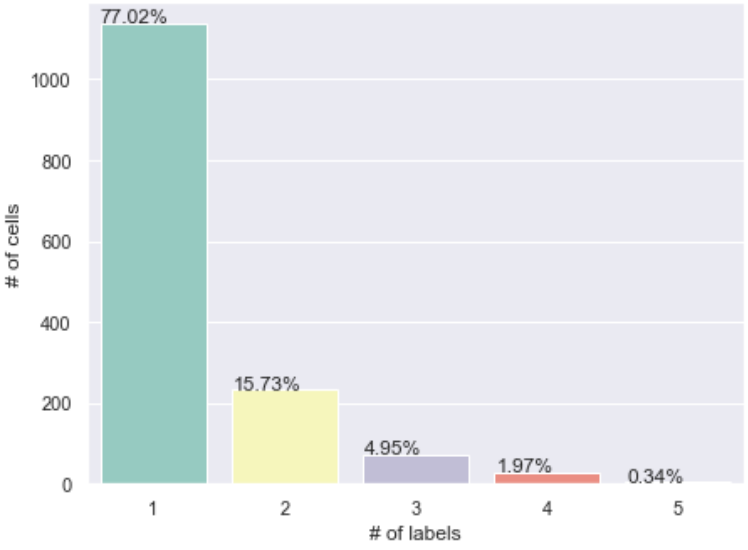

Figure 5.14: Number of classification labels per cell

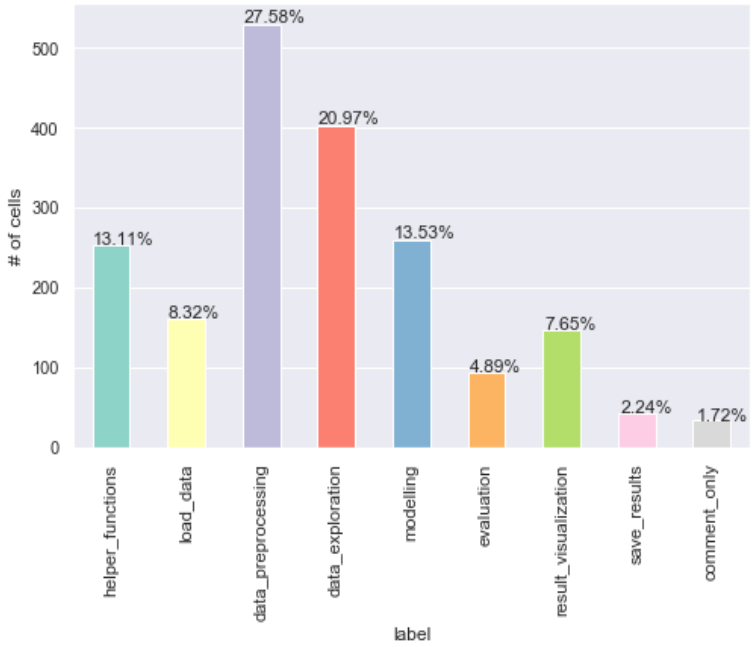

Figure 5.15: Number of cells per label

Figure 5.15 shows the composition of data science activities in notebooks. Around 27% of the cells perform data_preprocessing and around 20% of the cells perform data_exploration while one would expect all data science activities to have a more or less equal share. data_preprocessing accounts for around 1/5th of all data science activities (leaving out general labels) which is in line with common knowledge. The interesting finding is that prediction as a data science activity do not seem to be appearing in many notebooks having a share of around only 3%. I suspect this might be due to notebooks being an explorative workflow system rather than a system that supports production-ready software. Whether only individual data science practitioners and learners use the system and not organizations are yet to be ascertained.

Figure 5.16 and 5.17 shows the importance of lines_of_code and output_type of a cell in identifying the classification label of the cell respectively. Figure 5.16 shows that around 30% of the times when a cell has only one line of code, it is more assigned to label:helper_functions. This shows the user behaviour of writing import statements or other helper functions separately in a single cell.

Similarly, in Figure 5.17, we see that 65% of the times data_exploration is assigned to cells which have some output. The unexpected finding is that 45% of the times, cells with label:data_exploration has no output. While annotating, I observed that most of the Kaggle notebooks do not have executed results and only unexecuted form of notebook. This could be one of the reasons why data_exploration cells do not have any output in

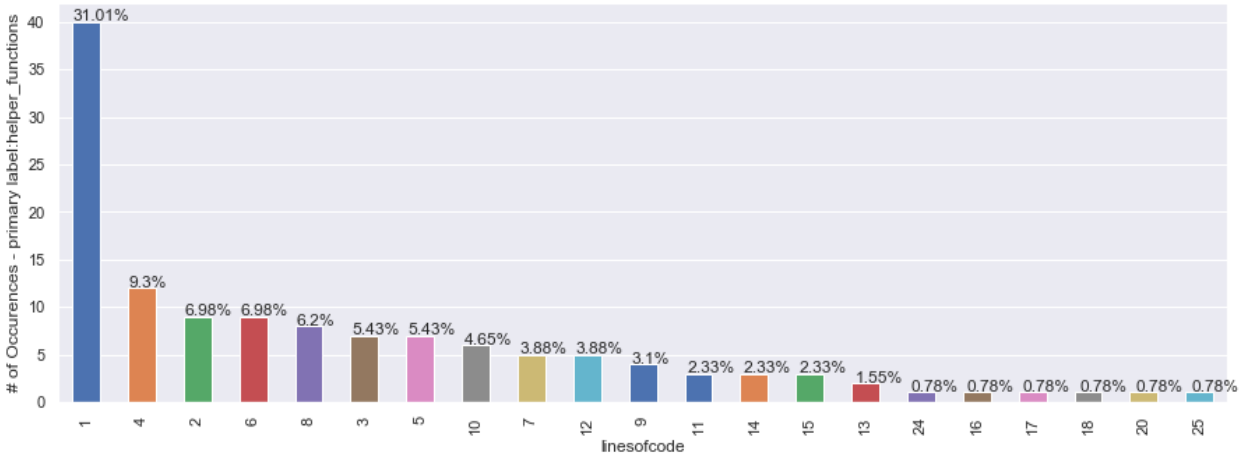

Figure 5.16: helper_functions vs. lines_of_code

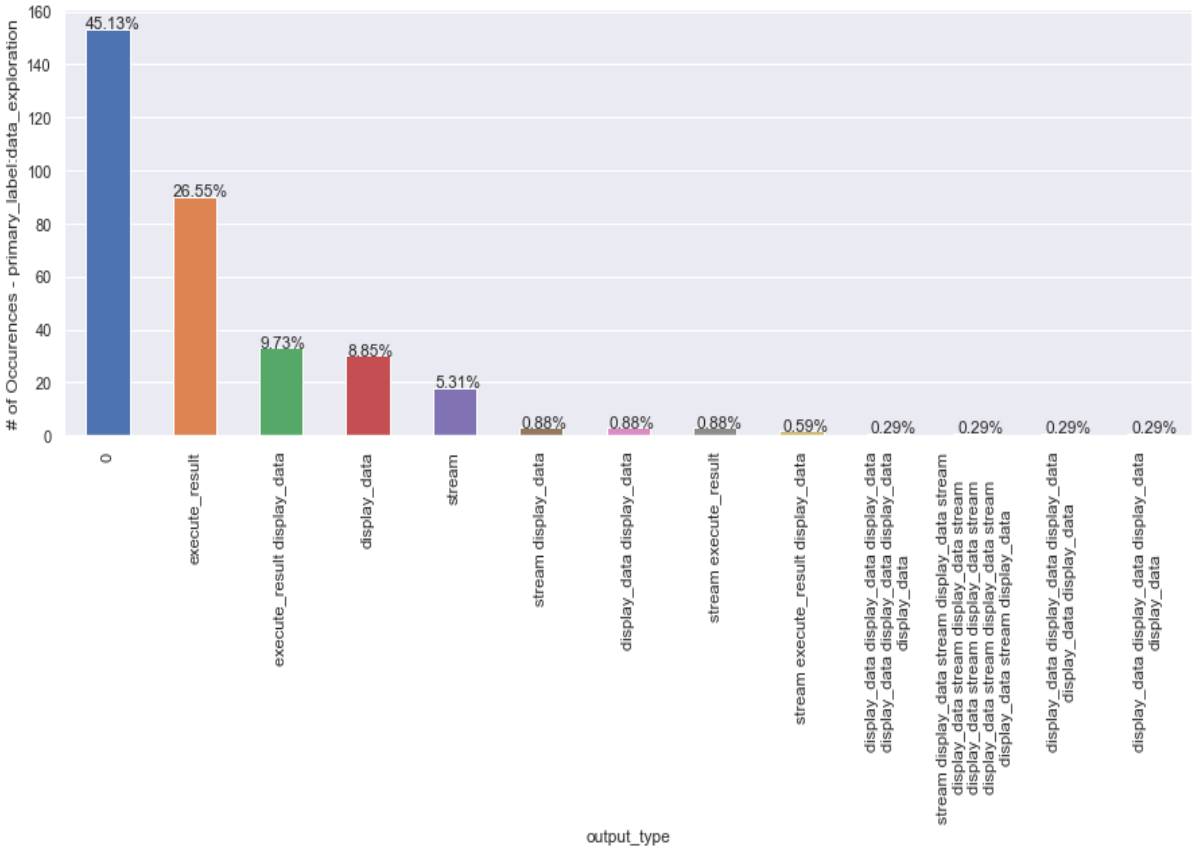

Figure 5.17: data_exploration vs. output_type

our dataset.

Lastly, I explore in brief wherein a notebook each of the data science activity labels appear. For this purpose, each notebook that has at least 9 code cells is divided into 9 parts/positions based on their cell_number ('comment_only' label is not considered). The positions and labels are: 1:helper_functions, 2:load_data, 3:data_preprocessing, 4:data_exploration, 5:modelling, 6:evaluation, 7:prediction, 8:result_visualization, 9:save_results. For example, if a notebook has 27 cells and the cell_number is 14, then the cell is considered to be in the 14/(27/9) = 5th part/position of the notebook and I ideally expect the 5th data science activity (modelling) in the cell. Similarly, I have computed the positions of the data science activities in the notebooks and visualized in Figure 5.18. It shows that helper_functions occur mostly at the beginning of the notebook while load_data, modelling, prediction, evaluation and result_visualization have more or less defined positions in the notebook. The two most important data science activities taking a majority share in a data science pipeline, data_preprocessing and data_exploration occur in almost all of the positions. That is, data_preprocessing and data_exploration do not confine themselves to the initial part of a data science pipeline and occur throughout the pipeline. According to my observation during annotation experiment, one of the major reasons for such a behaviour is that there are notebooks that perform exclusively data_exploration activity. For data_preprocessing label, the reason might be that a data scientist/programmer is performing an activity and probably going back and rechecking her data, make modifications and so on. This could also mean that an ETL activity does not really produce data in a format that is readily usable and requires more modifications to get the data in the desired way for a data_scientist.

5.4.2 Unsupervised learning

I employed three unsupervised learning techniques: Latent Dirichlet Allocation, KMeans Clustering and Agglomerative Clustering to understand the number of latent topics in the dataset. I performed three analyses based on the features used: ***code***, ***code-markdown-raw***, and ***import statements***. The features used for the three analyses are as follows:

code. This analysis uses lines of code as data for learning. Preprocessing of the data is done using *custom_ text_ processing* method from the Preprocessor class.

code-markdown-raw. This analysis uses lines of code and lines of markdown and raw cells (essentially, all the content from notebook) as data for learning. Preprocessing of the data is done using *custom_ text_ processing* method for code and *text_ processing* method for markdown and raw data.

import statements. This analysis uses only import statements from *code* cells as data for learning. Preprocessing of the data is done using *import_ text_ processing* method from the Preprocessor class.

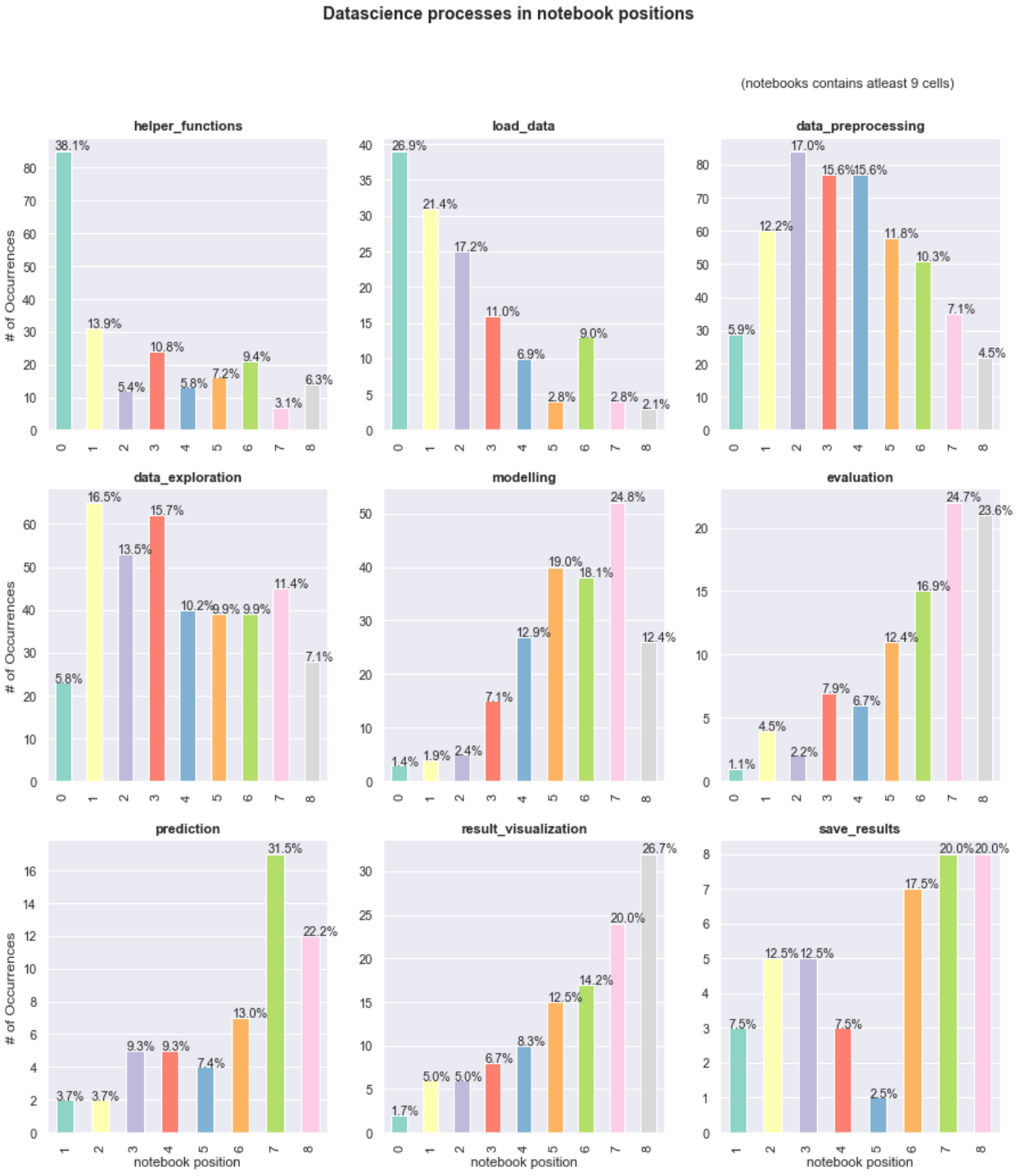

Figure 5.18: Position of labels in Notebooks

Vectorization of text based features is done using TfidfVectorizer[15]. I use uni, bi, tri-grams and do not exclude any features for the learning. Tf-idf has following parameter setup:

tfidf = TfidfVectorizer(ngram_range=(1,3), use_idf=True, stop_words='english')

The results for each method used are discussed below.

Results

LDA For topic modelling, ldamodel from gensim[16] is used and the parameters are as follows:

Lda(doc_term_matrix, num_topics=n, id2word=dictionary, passes=50, random_state=0)

Seed for num_topics are evaluated over *range(start=2, stop=10, step=2)*. Best number of topics are calculated based on the topic coherence_score (derived from CoherenceModel[17]).

Inference LDA topic modelling with all the features generated has predicted 8 topics but the topics are noisy and meaningless. As markdown contains a lot of task-specific data, I suspect it to be a likely source of the noise. With only code data (see Figure A.2), LDA has predicted 8 topics. The topics have distinguished, modelling and result_visualization activities, but still do not prove to be meaningful for our task. The noise in case of code only content could be coming from various user-generated variables. While LDA modelling with only import statements (see Figure A.3) has differentiated modelling activity, the predicted topics are just 2 and are not helpful in understanding the latent topics. See A.1, A.2, and A.3 for the figures from LDA topic modelling.

Agglomerative Clustering Agglomerative clustering uses the method AgglomerativeClustering from scikit-learn[18] and the parameters are as follows:

AgglomerativeClustering(n_clusters=n, affinity="euclidean", linkage="ward")

where n_clusters is defined based on Ward's minimum variance method[19].

[15] *https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html*

[16] *https://radimrehurek.com/gensim/models/ldamodel.html*

[17] *https://radimrehurek.com/gensim/models/coherencemodel.html*

[18] *https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html*

[19] *https://en.wikipedia.org/wiki/Ward%27s_method*
More information on implementation can be found in class:Clustering

Inference A total of 2 clusters were predicted for all the sets of features: code, code-markdown-raw, and import. The cophenet metric (distance between the clusters)[20] for import data was evaluated as 0.66 which was six times higher compared to that of code and code-markdown-raw, indicating clear boundaries between clusters. Whether the import data is enough to classify the points is indecisive from the results of Agglomerative Clustering and is dealt in one of the research questions (see Section 5.7.3). See A.4, A.5, and A.6 for the figures from Agglomerative clustering.

K Means KMeans clustering offered in sklearn[21] was used for this task. The parameters are setup as follows:

KMeans(n_clusters=n).fit(vectorized_corpus)

The clusters are visualized using PCA reduction[22] with n_components set to 3.

Seed for n_clusters parameter is set with the value:
mean(lda_predicted_number_of_topics, agglomerative_predicted_number_of_clusters)
of respective feature set (code, code-markdown-raw, import).

Inference KMeans clustering with code features predicted a total of 4 clusters. The clusters have some reasonable distinction (see Figure A.8) between helper_functions (numpy, pandas), modelling (x_train, y_train) and data_preprocessing. KMeans clustering of code-markdown-raw data and import data do not provide any useful results. See A.7, A.8, and A.9 for the figures from KMeans clustering.

Number of Topics From the inferences of all the three methods, the best number of clusters predicted vary around 4-8 and is in line with the number of classification labels that I have chosen for data science activities in the notebooks. It is also important to note that the analyses show unsupervised classification methods are not the best strategy for code classification of data science notebooks.

5.5 Code Cell Classification

In this section, I investigate Singlelabel Multiclass classification and Multilabel Multiclass classification for the code cell classification problem. For Singlelabel classification, I use primary_label as the classification label, which means each data point will have one label. For Multilabel classification, I have used all the labels applicable to a data point including primary_label. In Singlelabel and Multilabel classification, I first provide a

[20] *https://docs.scipy.org/doc/scipy-0.17.1/reference/generated/scipy.cluster.hierarchy.cophenet.html#scipy.cluster.hierarchy.cophenet*
[21] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
[22] *https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html*

comparison of the performance of various classifiers and then proceed to discuss the best performing classifier in detail. I also discuss in brief the performance of classifiers over four sets of features: all the features generated, code, code-markdown-comments, and import libraries. Please note that for evaluating import libraries, I replace the lines of code with the description of the library invoked. For example, if a cell contains *pd.read_csv('test.csv')*, I replace it with the man-page description of *pandas*. This helps in overcoming the sparsity of library tokens. The implementation of both Single-label Multiclass classification and Multilabel Multiclass classification uses scikit-learn implementation. A total of 1475 datapoints (code cells) are used for training using cross-validation (90% of the notebooks in the dataset). The test set contains a total of 173 datapoints (code cells) which is 10% of the notebooks in the dataset.

Before applying the classification model, I preprocess the text features in the dataset (training, validation, and test) using the cutom_text_preprocessing in the Class:Preprocessing. Machine learning methods require features to be represented as numeric. TfidfVectorizer method from scikit-learn[23] was used to transform the text features into numerics. The setup for TfidfVectorizer are as follows:

Singlelabel classification: *TfidfVectorizer(ngram_range=(1,3), use_idf=True, stop_words='english')*

Multilabel classification: *TfidfVectorizer(ngram_range=(1,3), use_idf=True, max_df=0.2, min_df=2, stop_words='english')*

Wherever applicable I have also applied categorization as a means for transforming text features: output_type, output_name, kernel_language, language, language_version, repo_id, filename, owner, r_cyclomatic_complexity_rank, r_maintainability_rank, primary_label. I used chi2 method for feature selection to select k most relevant features from the text features (without feature selection, the text features generally are around 120,000). I have retained all the other features generated for classification. The features are then standardized[24] to represent them in same scales before being passed to the classifier.

5.5.1 Singlelabel Classification

I evaluate a total of 9 classifiers for Singlelabel classification: Linear SVC, SVC, Logistic Regression, Random Forest, Decision Tree, Gradient Boosting, KNearest Neighbors, Multinomial Naive Bayes and Multilayer Perceptron. For each classifier, I estimate the parameters using GridSearchCV cross-validation and train them with the training dataset. Each of the trained classifiers is then evaluated against the test set and the performances are discussed below. Feature vector contains 2000 text features selected

[23] *https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html*

[24] *https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html*

model	all features	code	code-markdown-comments	import
mlp	0.509	0.543	0.572	0.306
linearsvc	0.434	0.613	0.532	0.289
kneighbors	0.422	0.520	0.538	0.214
gbclassifier	0.590	0.647	0.572	0.208
logistic	0.503	0.578	0.607	0.289
svc	0.468	0.162	0.566	0.162
randomforest	0.514	0.561	0.526	0.220
dt	0.393	0.538	0.509	0.197
multinomial_nb	NaN	0.509	NaN	0.266

Table 5.3: Comparison of accuracy of classifiers over different set of features - Singlelabel classification

using *chi2* method. Statistical features are included while evaluating all the features generated.

Comparison of Classifiers

Table 5.3 shows the accuracies of all the classifiers over a different set of features[25]. Gradient Boosting classifier trained with the code has the highest accuracy of 64.7%. comparison. LinearSVC trained with the code has an accuracy of 61.3% and is the second best performing classifier. I found that all the classifiers perform worse when trained with only import statements. The classifiers perform better with either *code* or *code-markdown-comments*. The accuracies of all the classifiers evaluated follow the equation given below in case of Singlelabel Multiclass classification:

$$code\ (or)\ code\_markdown\_comments > all\ features > import\ statements$$

Figure 5.19 visualizes the table 5.3.

Best performing classifier: Gradient Boosting I take the best performing Gradient Boosting classifier trained with code for further analysis. The parameters evaluated by GridSearchCV for Gradient Boosting classifier is shown in Figure 5.20. Figure 5.21 shows accuracy, f1score and classification_report for the classifier and Figure 5.22 shows the confusion matrix.

While most of the classification labels have precision above 0.6, 'data_preprocessing' has a low precision value of 0.39. In terms of recall, 'load_data', 'evaluation' and 'prediction' have recall values lesser than 0.4. The classifier performs well in terms of both precision and recall with labels: 'helper_functions', 'modelling' and 'data_exploration'. The

[25]Nan denotes the particular classifier is not evaluated for the given feature.

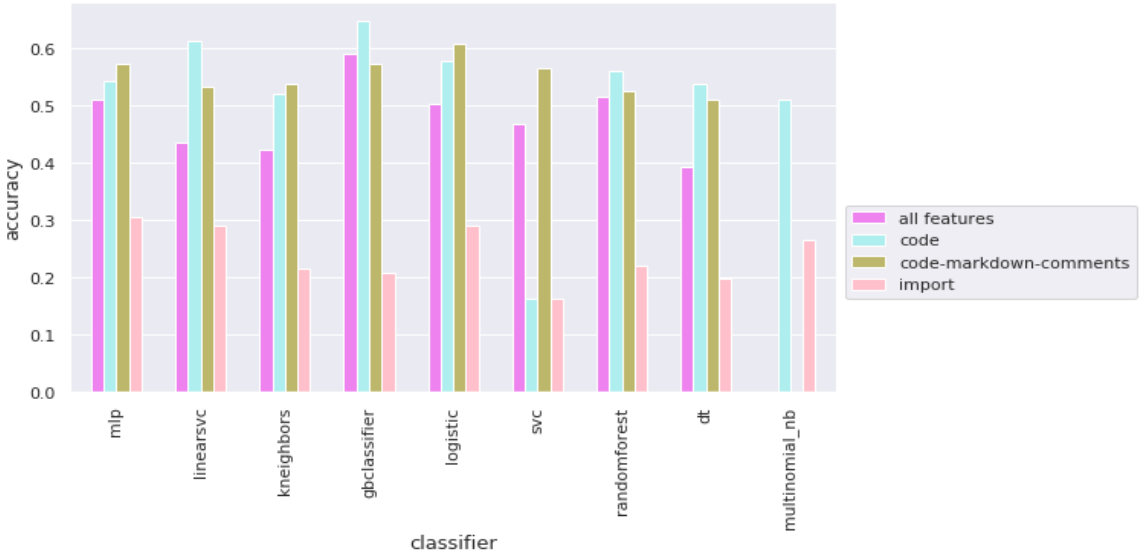

Figure 5.19: Classifiers Accuracy Comparison over Features - Singlelabel Classification

```
GradientBoostingClassifier(criterion='friedman_mse', init=None,
              learning_rate=0.1, loss='deviance', max_depth=3,
              max_features=None, max_leaf_nodes=None,
              min_impurity_decrease=0.0, min_impurity_split=None,
              min_samples_leaf=1, min_samples_split=2,
              min_weight_fraction_leaf=0.0, n_estimators=100,
              presort='auto', random_state=500, subsample=1.0, verbose=0,
              warm_start=False)
```

Figure 5.20: GradientBoosting Classifier - Singlelabel Classification - Parameters

```
Accuracy: 0.6473988439306358
F1 score: [0.85714286 0.47619048 0.50632911 0.69333333 0.84090909 0.36363636
 0.5        0.5        0.        ]
                      precision    recall  f1-score   support

    helper_functions       1.00      0.75      0.86        16
           load_data       0.71      0.36      0.48        14
  data_preprocessing       0.39      0.71      0.51        28
    data_exploration       0.70      0.68      0.69        38
           modelling       0.77      0.93      0.84        40
          evaluation       0.67      0.25      0.36         8
          prediction       1.00      0.33      0.50         6
result_visualization       0.62      0.42      0.50        19
        comment_only       0.00      0.00      0.00         4

         avg / total       0.68      0.65      0.64       173

Confusion matrix
[[12  1  1  0  2  0  0  0  0]
 [ 0  5  7  2  0  0  0  0  0]
 [ 0  1 20  3  2  0  0  2  0]
 [ 0  0 10 26  0  0  0  2  0]
 [ 0  0  2  0 37  0  0  1  0]
 [ 0  0  3  1  2  2  0  0  0]
 [ 0  0  1  0  3  0  2  0  0]
 [ 0  0  5  4  1  1  0  8  0]
 [ 0  0  2  1  1  0  0  0  0]]
```

Figure 5.21: GradientBoosting Classifier - Singlelabel Classification - Metrics

classifier classifies labels: 'load_data', 'evaluation', 'prediction' and 'result_visualization' with higher precision and lower recall score, which means while most of them are true positives, a lot of true positives were also missed out (refer to 5.22). I suspect this is because of the imbalance in the distribution of classification labels in the training data set. Poor precision and recall scores of 'comment_only' label is also a result of imbalanced class distribution in the data. At the same time, 'data_preprocessing' has a higher recall score but low precision. This shows that many of the labels are predicted as 'data_preprocessing' even when they are not. This behaviour is visualized clearly in the confusion matrix 5.22. A reason for this is that the structure of a piece of code that performs 'data_preprocessing' activity is generic and similar to the lines of code that performs any other data science activity. Another important reason is also that I have used only 'primary_label' for the classification and the presence of features relevant to other labels in a given data point might be a source of confusion for the classifier. Note that in the set of true labels for the test set, I do not have any data point assigned to label 'save_results'.

5.5.2 Multilabel Classification

I evaluate a total of 6 classifiers for Multilabel classification: Linear SVC, SVC, Logistic Regression, Random Forest, Decision Tree and Gradient Boosting. It is important to note that for Multilabel classification, all the relevant labels are taken into account, which means multiple labels can exist for each data point. The technique used for Multilabel classification is binary relevance[26] method which is implemented by OnevsRest in scikit-learn. Similar to Singlelabel classification, for each classifier I estimate the parameters using GridSearchCV cross-validation and train them with the training dataset. Feature vector contains 1000 text features selected using *chi2* method and statistical features are included while evaluating all the features generated. Each trained classifier is evaluated against the test set and the performance of the classifiers are discussed below.

Comparison of Classifiers

In this section, I produce a brief comparison of the performance of classifiers over a various set of features for Multilabel classification. I use three metrics for evauation: subset/exact accuracy[27], hamming loss[28] and Jaccard similarity[29].

Table 5.4 shows different metrics evaluated for the classifiers over various set of features. Logistic regression classifier trained with code has achieved a subset accuray of 36.99%. Since I use a total of 10 classification labels in Multilabel classification, I expected the subset accuracy to be low as it is a harsh metric. Table 5.4 shows that all of the classifiers

[26] Binary relevance (OnevsRest in Scikit) *https://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html*
[27] *https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html*
[28] *https://scikit-learn.org/stable/modules/generated/sklearn.metrics.hamming_loss.html*
[29] *https://scikit-learn.org/stable/modules/generated/sklearn.metrics.jaccard_similarity_score.html*

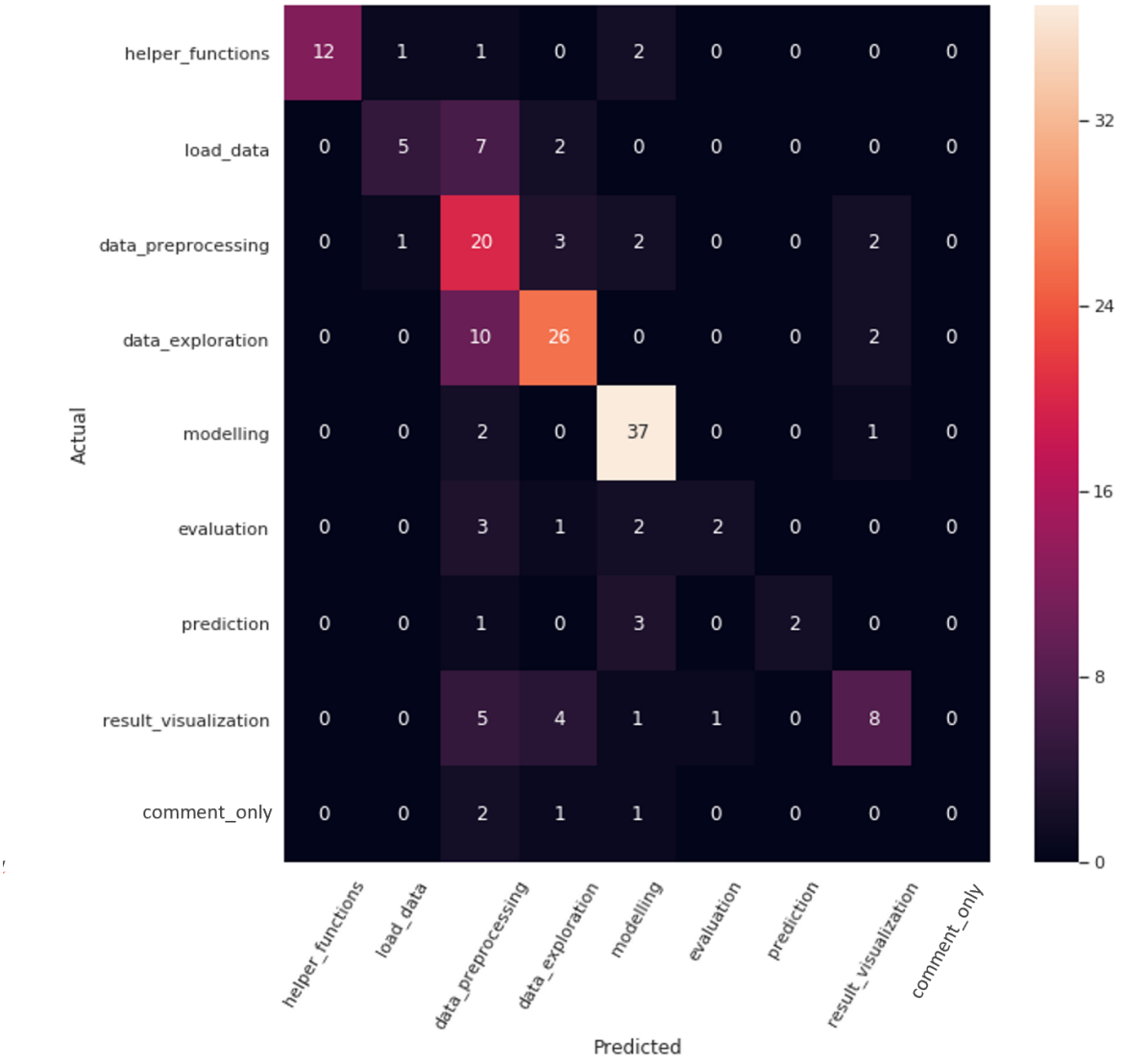

Figure 5.22: GradientBoosting Classifier - Singlelabel Classification - Confusion Matrix

model[feature set]	subset_acc	hamming_loss	jaccard_sim
rf[all features]	0.236994	0.115029	0.307611
gb[all features]	0.248555	0.115029	0.366763
dt[all features]	0.132948	0.179191	0.309345
svc[all features]	0.000000	0.147399	0.000000
lsvc[all features]	0.052023	0.239884	0.137187
log[all features]	0.219653	0.193642	0.359593
rf[code]	0.260116	0.120231	0.324952
gb[code]	0.225434	0.108671	0.358863
dt[code]	0.242775	0.142775	0.376879
svc[code]	0.000000	0.147399	0.000000
lsvc[code]	0.312139	0.107514	0.394220
log[code]	0.369942	0.108092	0.443353
rf[code-markdown-comments]	0.179191	0.123121	0.235645
gb[code-markdown-comments]	0.260116	0.117919	0.334586
dt[code-markdown-comments]	0.150289	0.136994	0.326686
svc[code-markdown-comments]	0.000000	0.147399	0.000000
lsvc[code-markdown-comments]	0.306358	0.118497	0.396050
log[code-markdown-comments]	0.294798	0.124277	0.349326
rf[import]	0.109827	0.136416	0.201252
gb[import]	0.132948	0.138728	0.210019
dt[import]	0.098266	0.144509	0.199904
svc[import]	0.000000	0.147399	0.000000
lsvc[import]	0.069364	0.145665	0.141618
log[import]	0.132948	0.142197	0.190173

Table 5.4: Comparison of metrics of classifiers over different set of features - Multilabel classification

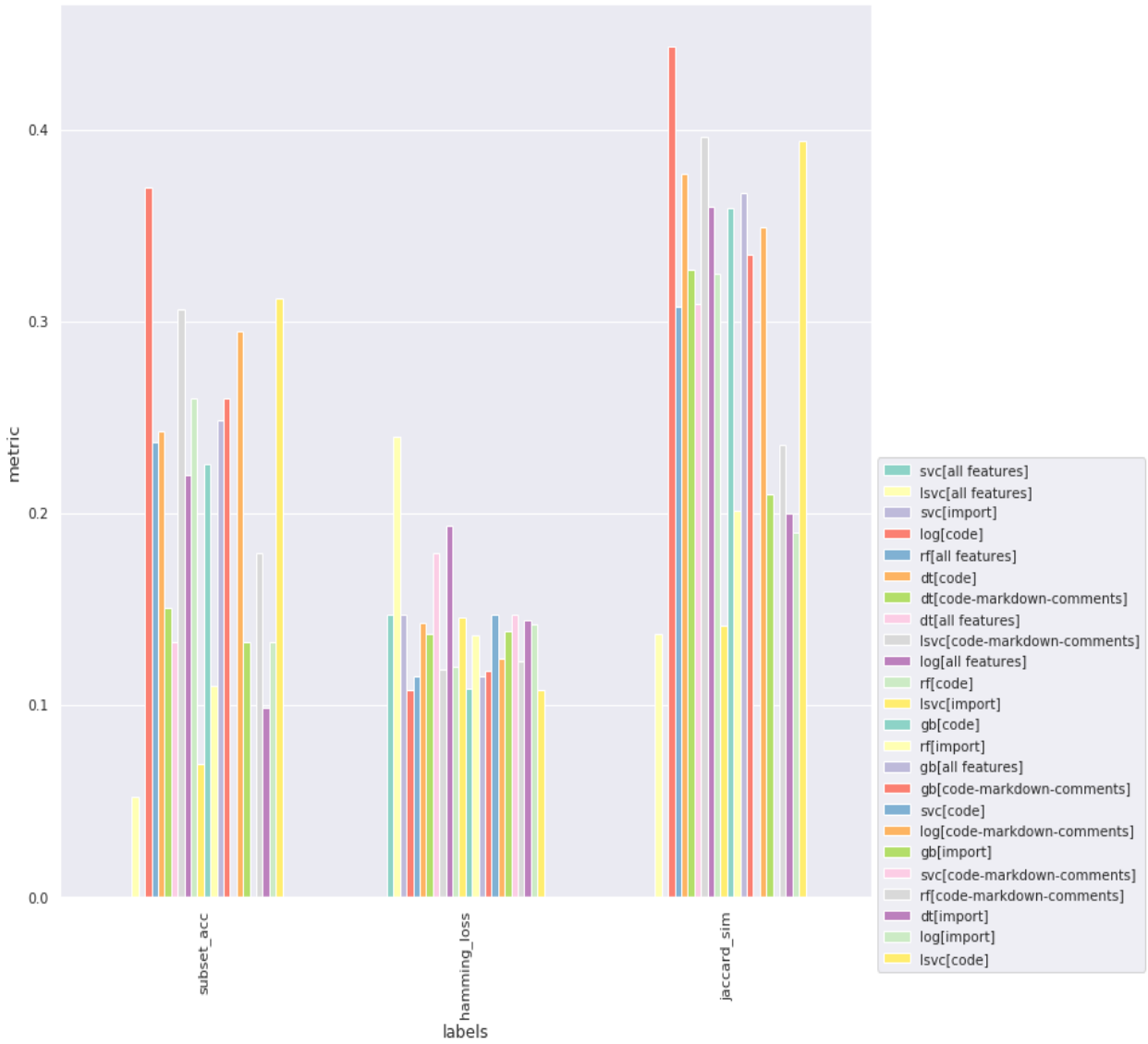

Figure 5.23: Comparison of metrics of classifiers over different set of features - Multilabel classification

have a hamming loss that is <20%. A low hamming loss score means that the proportion of labels predicted incorrectly is low. I observed that many of the data points were classified as either one of the classification labels or none (see Figure 5.25). The low hamming scores are misleading because the composition of labels varies hugely in the set of true labels (see Figure 5.24) compared to the set of predicted labels. Subset accuracy, a harsh metric, also do not seem appropriate for the classification task since I have more classes. Jaccard similarity is a midpoint between hamming loss and subset accuracy [A. F. Park and Read, 2019] and is more likely to show a balanced view of the classifier's performance. It measures the similarity between the set of predicted labels and the corresponding set of true labels. Logistic regression classifier trained with code performs the best in terms of Jaccard similarity coefficient score achieving 44.3%. This means that the set of prediction labels prediction by logistic regression classifier is ~45% similar to the set of true labels. We can also see from the table 5.4 that the second best performing classifier is Linear SVC with a Jaccard similarity score of 39.4%. Figure 5.23 visualizes the table 5.4.

Best performing classifier: Logistic Regression I take the best performing logistic regression classifier trained with code for further analysis. The parameters evaluated by GridSearchCV for logistic regression classifier is shown in Figure 5.26. Figure 5.27 shows f1score and classification_report for the classifier and Figure 5.28 shows the receiver operating characteristic curve for logistic regression classifier.

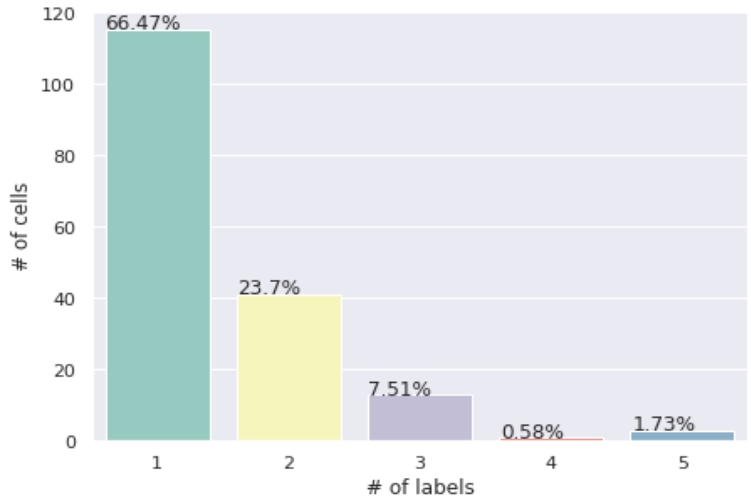

Figure 5.24: True label set composition

Most of the classification labels have precision value above 0.5 except 'prediction' which has a precision of 0.33. In terms of recall, 'helper_functions', 'modelling' and 'result_visualization' have recall values higher than 0.55. This means the classifier performs well in terms of both precision and recall with labels: 'helper_functions', 'modelling' and 'result_visualization'. The classifier classifies all the labels with high precision and low recall score except 'save_results' and 'comment_only', both of which have precision and recall scores of 0. This indicates that while true positives have improved, false negatives have also increased. I suspect, an imbalance in the distribution of classificiation labels is the reason for an increased number of false negatives. While I use all the relevant labels for a given point, the data set still has a skewed distribution of classes. Also, since the binary relevance method trains a binary classifier for each of the classes, I am unable to leverage any possible correlations between labels.

Figure 5.29 shows the accuracy score for each label in a binary relevance method using logistic regression classifier. This is in line with the observation of a low hamming loss. Figure 5.24 and 5.25 shows that the logistic regression classifier do not assign any label to a massive 30% of the data points. While this is a concern which is reflected in a low recall score, it also means not many labels are inaccurately assigned.

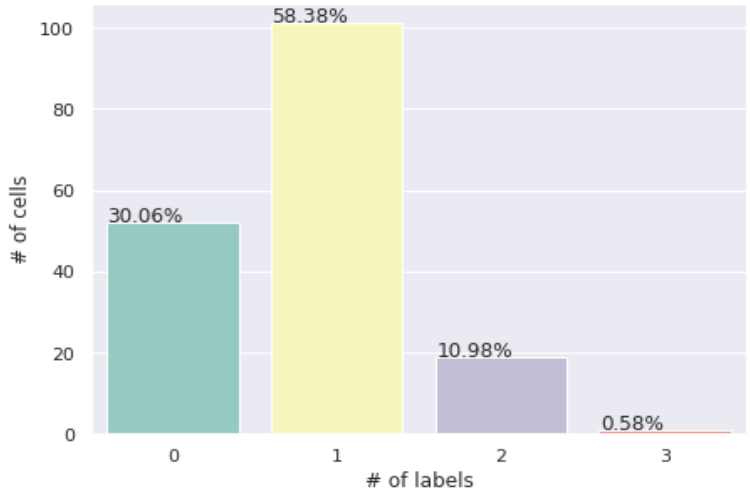

Figure 5.25: Predicted label set composition - Logistic Regression Classifier

```
GridSearchCV(cv=PredefinedSplit(test_fold=array([-1, -1, ...,  5,  5])),
       error_score='raise',
       estimator=OneVsRestClassifier(estimator=Pipeline(memory=None,
     steps=[('chi2', SelectKBest(k=1000, score_func=<function chi2 at 0x7f2b57f72488>)), ('log', LogisticRegression(C=1.0, cl
ass_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=500, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False))]),
          n_jobs=1),
       fit_params=None, iid=True, n_jobs=1,
       param_grid={'estimator__log__solver': ['newton-cg', 'sag', 'lbfgs'], 'estimator__log__C': array([ 0.1,  1. , 10. ]),
'estimator__log__penalty': ['l2'], 'estimator__log__multi_class': ['ovr']},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='accuracy', verbose=0)
```

Figure 5.26: Logistic Regression Classifier - Multilabel Classification - Parameters

```
Multilabel classification:
F1 score: [0.84       0.26315789 0.43037975 0.15686275 0.86419753 0.70967742
 0.14285714 0.44897959 0.         0.        ]
                     precision    recall  f1-score   support

    helper_functions       0.95      0.75      0.84        28
           load_data       1.00      0.15      0.26        33
  data_preprocessing       0.52      0.37      0.43        46
    data_exploration       0.50      0.09      0.16        43
           modelling       0.90      0.83      0.86        42
          evaluation       1.00      0.55      0.71        20
          prediction       0.33      0.09      0.14        11
result_visualization       0.52      0.39      0.45        28
        save_results       0.00      0.00      0.00         0
        comment_only       0.00      0.00      0.00         4

         avg / total       0.71      0.41      0.48       255
```

Figure 5.27: Logistic Regression Classifier - Multilabel Classification - Metrics

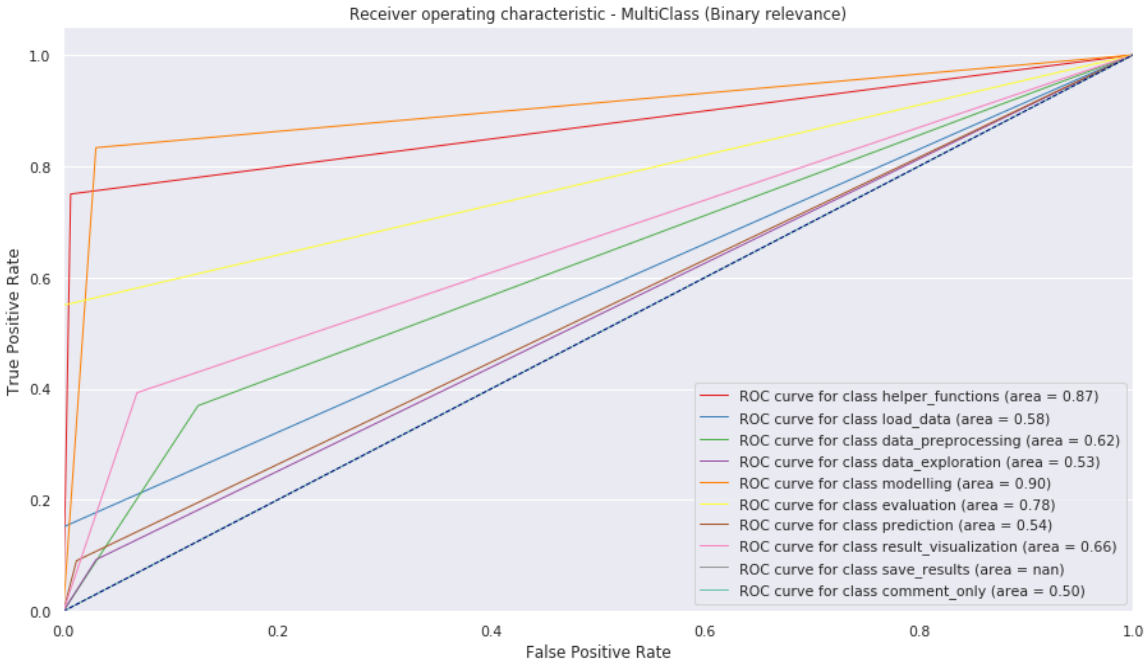

Figure 5.28: Logistic Regression Classifier - Multilabel Classification - ROC

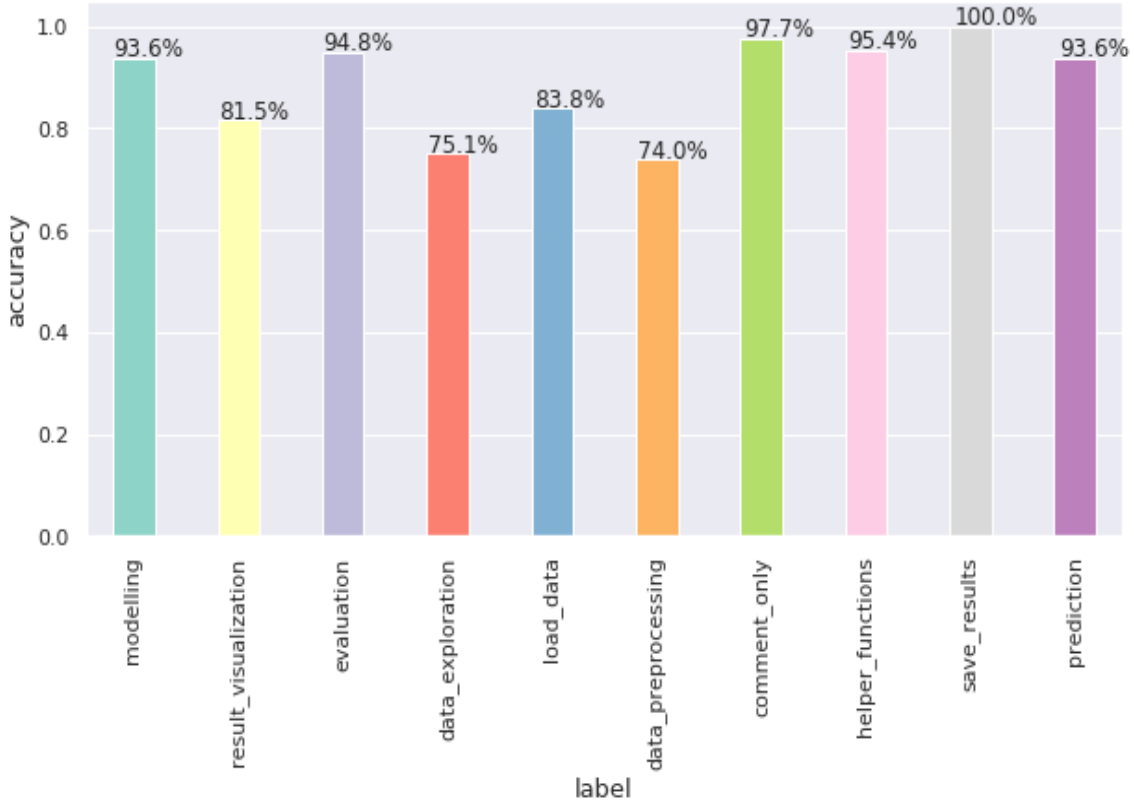

Figure 5.29: Logistic Regression Classifier - Multilabel Classification - Accuracies

```
<no:Code rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_60623_8">
  <no:type>code</no:type>
  <no:hasDataScienceActivity rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Evaluation"/>
  <no:isNotebookCellOf rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_60623"/>
  <no:hasDataScienceActivity rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Modelling"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
</no:Code>
```

Figure 5.30: Notebook Cell Object respresentation in RDF format using Data Science Process and Notebook Ontology

5.5.3 Evaluation and Prediction

To have an automatic annotation system that produces accurate annotations, the classifier should ideally produce zero false positives. A false positive is costly because an inaccurately annotated data point is worse than an unannotated datapoint. With that goal, Multilabel classification is preferred over Singlelabel classification for automatic annotation system based on the average precision results. While Singlelabel classification has a higher average recall value compared to Multilabel classification, the trade-off is to have a system that produces annotation with high precision. Hence, I use Multilabel classification with logistic regression classifier to predict the labels for the test set.

5.6 Annotated Dataset as Output

5.6.1 RDF Serialization

I use the information from test data and the corresponding labels predicted to produce a data set containing RDF annotations. Data Science Process and Notebook Ontology (refer to 4.1.2), denoted as *'no'* is used as the ontology for annotation. Figure 5.30 and 5.31 illustrates the annotations serialized in RDF.

Figure 5.30 shows the RDF serialization of a single datapoint, a *code* cell. Name of the notebook cell is identified with <filename>_<cell_number>. *no#nb_60623_8* is a NamedIndividual of no:type *code*, no:isNotebookCellOf *no#nb_60623*, contains no:hasDataScienceActivity *no#Evaluation* and no:hasDataScienceActivity *no#Modelling*.

Figure 5.31 shows the RDF serialization of a single notebook object. *no#nb_111963* is a NamedIndividual of no:type *Notebook*, has no:name *nb_111963* and no:owner *1596037*, written in no:programming_language *python*. It is no:published_in_platform *github*, developed in no:workflow_system *jupyter* and has no:file_extension *.ipynb*. no:url indicates where the notebook resides. It also contains no:hasNotebookCell *no#nb_111963_0* which has no:hasDataScienceActivity *no#Modelling*.

```
<no:Notebook rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777">
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_11"/>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_23"/>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_1"/>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_17"/>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_8"/>
  <no:file_extension></no:file_extension>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_9"/>
  <no:published_in_platform>github</no:published_in_platform>
  <no:programming_language>python</no:programming_language>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_21"/>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_20"/>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_6"/>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_18"/>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_19"/>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_5"/>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_4"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_10"/>
  <no:workflow_system>jupyter</no:workflow_system>
  <no:workflow_type>datascience</no:workflow_type>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_16"/>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_0"/>
  <no:owner>11899312</no:owner>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_15"/>
  <no:url>https://github.com/mikkawi/P5-Submission/blob/fd924b3ee7324d7b3a9ed621457c0d63bdd8d650/
                                    P5-Vehicle_detection-Exploration-ver2.ipynb</no:url>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_3"/>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_2"/>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_14"/>
  <no:name>nb_57777</no:name>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_22"/>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_7"/>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_13"/>
  <no:hasNotebookCell rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_57777_12"/>
</no:Notebook>
```

Figure 5.31: Notebook Object respresentation in RDF format using Data Science Process and Notebook Ontology

```
q1 = g1.query("""SELECT ?file ?title ?cell WHERE{?file no:name ?title.
?file no:hasNotebookCell ?cell.
?cell no:hasDataScienceActivity no:Prediction}""")
```

Figure 5.32: A simple SPARQL query

5.6.2 SPARQL Analysis

SPARQL (SPARQL Protocol and RDF Query Language) is a semantic query language for databases, "able to retrieve and manipulate data stored in Resource Description Framework (RDF) format"[30].

The RDF format of the output from the classification task allows its to be queried by SPARQL queries. A simple SPARQL query on the dataset containing RDF annotations using library rdflib[31] is demonstrated in the Figure 5.32. It queries for notebooks containing a cell that performs data science activity 'prediction'. The query produces the following result:

3 results
name: http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_37187
title: nb_37187
cell: http://www.semanticweb.org/ramas/ontologies/2019/0/no#nb_37187_19

name: http://www.semanticweb.org/ramas/ontologies/2019/0/no#caravan-insurance-customer-identification
title: caravan-insurance-customer-identification
cell: http://www.semanticweb.org/ramas/ontologies/2019/0/no#caravan-insurance-customer-identification_28

name: http://www.semanticweb.org/ramas/ontologies/2019/0/no#caravan-insurance-customer-identification
title: caravan-insurance-customer-identification
cell: http://www.semanticweb.org/ramas/ontologies/2019/0/no#caravan-insurance-customer-identification_20

Thus, the semantic web-based dataset containing RDF annotations allows us to take advantage of the semantics of the notebooks and data science activities in them for further analyses in the area of code discovery and other machine learning problems.

[30] *https://en.wikipedia.org/wiki/SPARQL*
[31] *https://rdflib.readthedocs.io/en/stable/*

5.7 Answers to the research questions

I have drawn the conclusions to the research questions based on the evaluation of various classifiers on different feature sets.

5.7.1 What features in the notebook are more informatory to the automatic classification of data science notebooks?

I have discussed this in Section 6. In general, I have found that the code features generated are more informatory than other features in the notebooks for classification.

5.7.2 Do non-code features like markdown/comments in notebooks improve classification accuracy?

No. I found that in both Singlelabel and Multilabel classification, code features give better accuracy leading to the conclusion that non-code features like markdown/comments do not really improve the classification accuracy.

5.7.3 Are import statements along with their library functions sufficient to classify the code according to their data science steps?

No. I found that in both Singlelabel and Multilabel classification, classifiers trained with only import statements along with the library functions have always performed significantly worse than all of the other features sets I evaluated.

5.7.4 Do popular coders produce notebooks that are easier to classify/have higher classification accuracy?

Inconclusive. I have tested whether notebooks with higher star_count or higher fork_count has better accuracy than the ones with lower star_count or lower fork_count respectively. The accuracy for each label is obtained using the binary relevance method of Multilabel classification using logistic regression classifier (best performing classifier). I split the notebooks into popular and unpopular (50%:50%) set based on their fork_count or start_count and computed scikit-learn accuracy_score. The results are inconclusive since for some of the classes, popular notebooks have higher classification accuracy while for the others, unpopular notebooks have higher classification accuracy.

6

Discussions

In this chapter, I discuss more on the performance of the classifiers for code cell classification of notebooks.

6.1 Singlelabel Classification

feature_importances_. I used feature_importances_[1] attribute from DecisionTreeRegressor to understand the importance of features. Feature_importances_ are calculated based on how much each feature decreases the weighted impurity of the forest and is a good measure to understand the importance of features in a machine learning task.

I found the top 30 features from one of the DecisionTreeRegressors available in the best performing Singlelabel classifier, Gradient Boosting classifier. They are: read_csv, open, algebra import, pd read_sql, datetime, values, seaborn sns matplotlib, zip tqdm_notebook range, fpr tpr, full_protein_model, fs, format test_accuracy, fpr, get_accuracy data, format path, format datetime pred, format datetime, forest_fit, full_protein_model h5, get_color_pal, get_accuracy data bias, forest confusion_matrix, get_feature_names to_pickle, get_feature_names to_pickle term_list, get_tensor_by_name, get_train_data, get_transforms, graph_b, graph_w, green, green alpha, green alpha plt, green_path, forest confusion_matrix cv_target, foreign_keys cursor, forest, foreign_keys cursor db, file pd, file pd read_csv, file_ids, filename labelled, filename labelled heatmap, filter_movies, filter_movies index, finalproject, finalproject db, finalproject db db, fit, fit x_train, fit x_train y_train.

statistical features. In a classification task, when the feature vector includes both text-based features and statistical features, some of the statistical features have been found to have high importance. In the case of Singlelabel classification using Gradient Boosting classifier, two of the top ten features are statistical features. They are r_sloc and variable_count. At the same time, in the Singlelabel classification using Random Forest

[1] feature_importances_ *https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html*

classifier, six of the top ten features by their importance are statistical features. They are cell_number, variable_count, linesofcode, execution_count, code_tokens_per_nb and output_type.

Representativeness of the dataset. I also analysed whether the performance of the classifier using Singlelabel classification varied hugely if a different training and test dataset were to be used. I found that the performance varied +/- 5% depending on the classifier. This shows that the dataset is fairly representative of the jupyter notebook data. However, having more data points to learn from will make the classifier more robust.

6.2 Multilabel Classification

Classification labels in test and predicted set. Figure 6.1 shows the true composition of the classification labels in the test set and Figure 6.2 shows the composition of the predicted labels using Multilabel classification. They show that 'load_data' and 'data_exploration' are the highly misclassified labels. While more datapoints are classified as 'data_preprocessing' as expected, 6.2 shows that even more data points than expected are classified as 'modelling'. The next highly represented labels are ' helper_functions' and 'result_visualization' which have representation around ~5% higher than their true representation. I suspect the higher representation of 'result_visualization' and lower representation of 'data_exploration' is related since they have very similar features. Data points misclassified as 'modelling' are shown below and it clearly reveals that the two important reasons for misclassification are: one, lack of distinct structure for certain data science activities and two, lack of enough data points in the training dataset to learn from.

Features : [y_pred clf.predict x_test print metrics.accuracy_score y_test y_pred helper_functions]
True labels: 'evaluation', 'prediction'
Predicted labels: 'modelling', 'evaluation', 'prediction'

Features: [clf clf_nb helper_functions]
True label: 'prediction'
Predicted label: 'modelling'

Size of the feature vector. I analysed whether the performance of the classifier varied given the size of the feature vector. I found that there is no significant pattern in the performance of the classifier depending on the size of the feature vector used (see Figure 6.3). This shows that increasing the feature size does not necessarily mean an increase in the performance of the classifier. I also tested how the classifier performed when the preprocessed code did not contain uninformative features like user-generated variables. While this did not improve the performance, it helps in reducing feature size and keep the model simple to draw an inference.

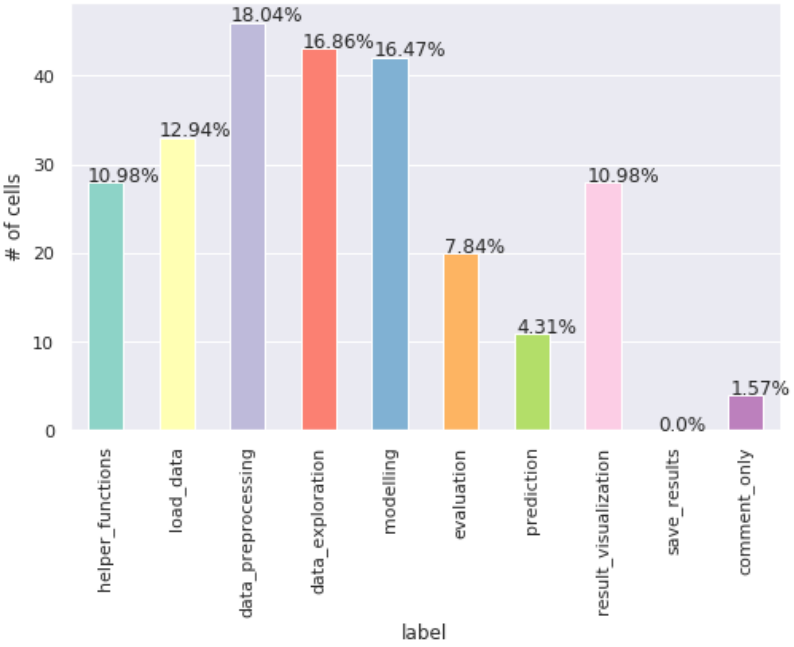

Figure 6.1: Labels in test set

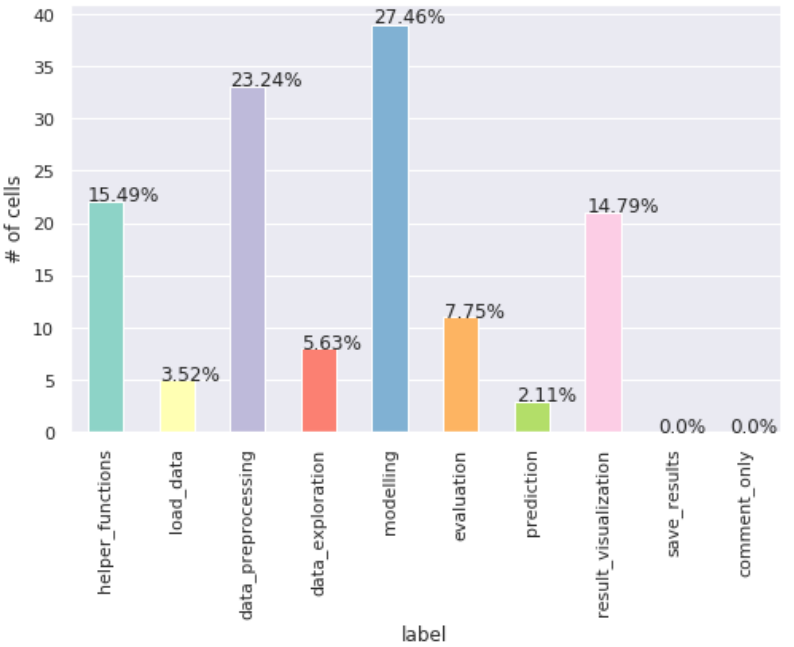

Figure 6.2: Labels in predicted set

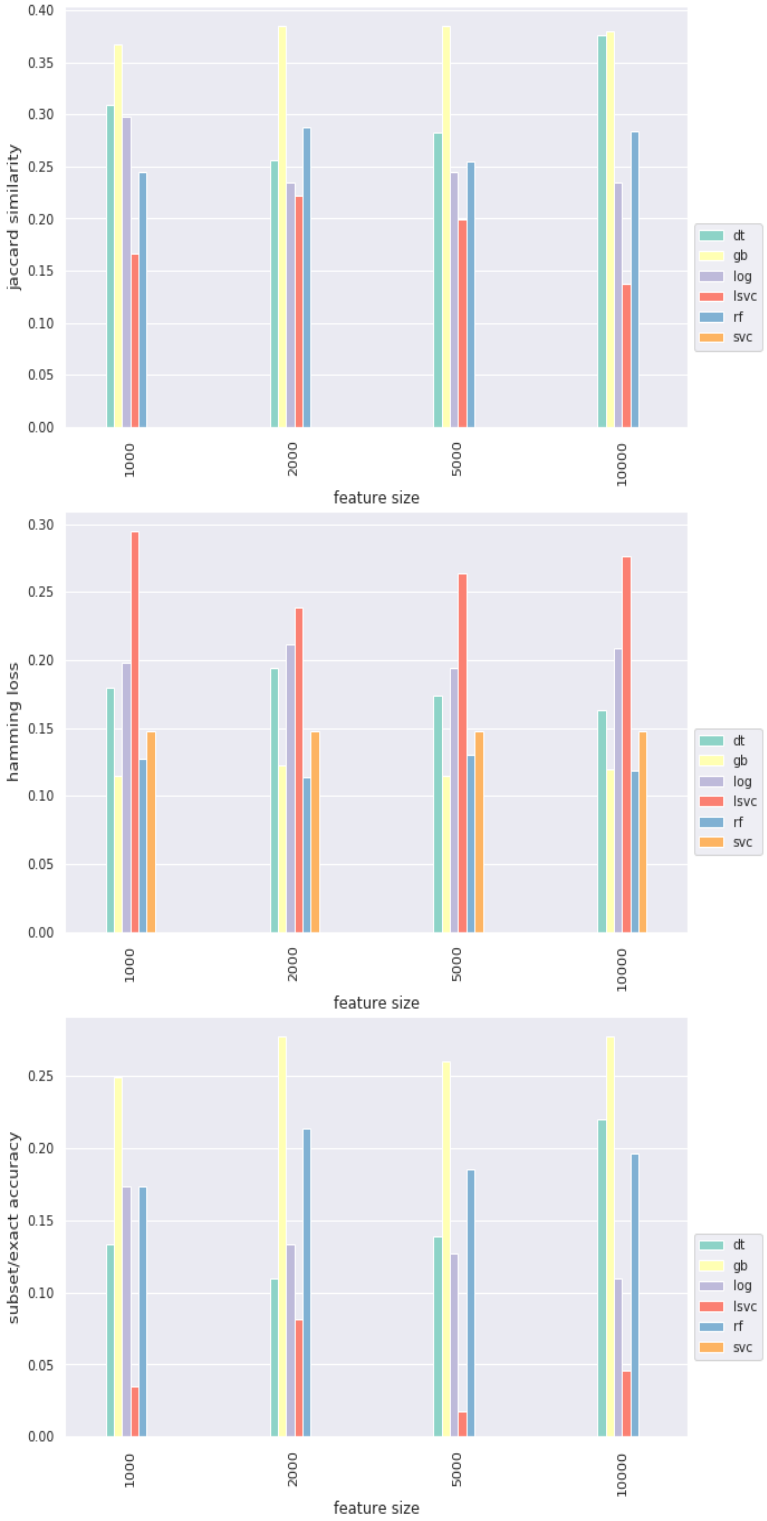

Figure 6.3: Evaluation Metrics - Multilabel Classification - Size of the feature vector

6.3 General Comments

Distribution of classification labels. The distribution of classes in the dataset is skewed towards some classes. This affects the performance of classifiers as discussed in 5.5. While Multilabel classification reduces this bias by taking into account all of the relevant labels, thereby increasing data points for each class, a larger training dataset will further mitigate the bias raising from an imbalanced distribution of classification labels. While I do believe that an increase in training dataset size will improve the performance of both Singlelabel and Multilabel classification, it is difficult to achieve a balanced class distribution in case of data science activities. This is because of the fact that some data science activities like 'data_preprocessing' happen more often than other labels and are coded generally in multiple cells to implement their functionalities. Nevertheless, to the make system more robust to classification labels that are inherently difficult to classify, it is important to have enough representation of them.

Inter-annotator agreement. The low Cohen-kappa scores, in general, show the ambiguity in identifying data science activities in a data science pipeline. This shows that data science activities do not have universal agreement on their definitions and practices and have a lot of room for improvement. This reaffirms the thesis's motivation to design methodologies and frameworks that enable a high-quality data science.

7

Conclusions

This thesis is an endeavour devoted to a topic which is being increasingly adopted by various discipline: Data science. Understanding the way data science is designed, implemented and reported by data scientists is important to provide tools and frameworks that enable high-quality data science. As a first step, I have implemented an automatic annotation system to classify the data science notebooks according to the data science activity they perform. This has enhanced my understanding of data science steps and its overlapping nature along with the software engineering aspects (readability, production-readiness etc.) of notebooks in general. Whereas previous methods have focused on source code classification in general, considerable progress has been made in this thesis about the classification of code in notebooks implemented using an interactive platform, i.e., Jupyter which do not follow the same coding standards as a normal piece of code in IDEs.

For this thesis, I have generated and discussed various features for the notebook classification. I also explored the latent topics in the notebooks along three lines: code, code-markdown-raw and import statements using unsupervised learning methods such as LDA Topic Modelling, Agglomerative Clustering and KMeans Clustering. The analyses also show that, given the data set, unsupervised learning methods do not provide useful results in understanding the latent topics. I also have elaborated on the exploratory analysis of the dataset extensively. For classification, I have analysed both Singlelabel Multiclass classification and Multilabel Multiclass classification and discussed the advantages of applying one over the other. The analysis led to the conclusion that Multilabel classification using logistic regression applied over code features has a high precision. Further analysis into classification labels and feature importance have provided an insight into the steps of the data science pipeline and exposed their non-sequential nature in practice. Additionally, I have also evaluated the classifiers with respect to a various set of features in order to answer the research questions on the relevance of meta-features in a code classification system.
The dataset generated with RDF annotations using the ontology Data Science Process and Notebook proves sufficient to represent the notebooks and its data science activities. I also demonstrated, in brief, the capability of the annotated dataset using a simple

SPARQL query.

This thesis has provided the first comprehensive code classification for notebooks according to the data science activities and code classification for notebooks in general[1]. I have discussed various features, classification methods, classifiers and their performances. The present findings of the occurrence of data science steps in notebooks have important implications for improving the data science pipeline. I hope that this work will be beneficial in code classification and semantic analysis of notebooks and also to understand the data science implementation in practice. It is also important to note that the automatic annotation method for data science notebooks could be extended to annotations of other types given an appropriate set of classification labels whereas the feature set presented can be used in machine learning methods focused towards various tasks.

[1] to the best of our knowledge

8

Future Work

The thesis provides a lot of scopes to take the work further in its endeavour to provide high-quality data science. It was out of the scope of this thesis to test how the models would perform if word embeddings or code embeddings representation is used instead of Tfidf representation for text-based features. Further research on this should be undertaken. I believe more analysis into the features and classification of data science notebooks with regards to GitHub and Kaggle respectively could reveal more about the users who implement data science tasks and their characteristics. Automatic feature discovery of notebooks would be an area worth exploring as well. It remains also to be tested whether deep learning networks like CNN (which has worked well on text data) and LSTM (sequence prediction) will improve the classification accuracy of notebook code cell classification. Furthermore, I also plan to explore an active learning approach, to make the algorithm select the notebooks to be labelled to reduce the cost of expert labelling. These are reserved for future work.

As a next step, I plan to implement a jupyter plugin using the annotation system which will enable me to both get annotation from more users and also provide guidance into what data science step the user should take next in their data science task. I also plan to review and extend the Data Science Process and Notebook Ontology by adding more properties, relationships and objects and also integrate them with other existing data science ontologies.

Last but not least, I plan to extend the ground truth, in order to have more samples labelled that will help the system in providing annotations that are more accurate.

References

[A. F. Park and Read, 2019] A. F. Park, L. and Read, J. (2019). *A Blended Metric for Multi-label Optimisation and Evaluation: Recognizing Outstanding Ph.D. Research*, pages 719–734.

[Altman, 1990] Altman, D. G. (1990). *Practical statistics for medical research*. Chapman and Hall/CRC.

[Bacchelli et al., 2012] Bacchelli, A., Dal Sasso, T., D'Ambros, M., and Lanza, M. (2012). Content classification of development emails. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 375–385, Piscataway, NJ, USA. IEEE Press.

[Barstad et al., 2014] Barstad, V., Goodwin, M., and Gjøsæter, T. (2014). Predicting source code quality with static analysis and machine learning.

[Binkley, 2007] Binkley, D. (2007). Source code analysis: A road map. In *IN FUTURE OF SOFTWARE ENGINEERING*, pages 104–119.

[Bonaccorso, 2018] Bonaccorso, G. (2018). *Machine Learning Algorithms - Second Edition*. Packt.

[Burges, 1998] Burges, C. J. C. (1998). A tutorial on support vector machines for pattern recognition. *Data Min. Knowl. Discov.*, 2(2):121–167.

[Cavnar and Trenkle, 1994] Cavnar, W. B. and Trenkle, J. M. (1994). N-gram-based text categorization. In *In Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175.

[Dhar, 2013] Dhar, V. (2013). Data science and prediction. *Commun. ACM*, 56(12):64–73.

[E. Schapire, 2002] E. Schapire, R. (2002). The boosting approach to machine learning: An overview. *Nonlin. Estimat. Classif. Lect. Notes Stat*, 171.

[E. Schapire, 2013] E. Schapire, R. (2013). *Explaining AdaBoost*, pages 37–52.

[Friedman, 2001] Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *Ann. Statist.*, 29(5):1189–1232.

[Gil et al., 2011] Gil, Y., Ratnakar, V., Kim, J., Gonzalez-Calero, P., Groth, P., Moody, J., and Deelman, E. (2011). Wings: Intelligent workflow-based design of computational experiments. *IEEE Intelligent Systems*, 26(1):62–72.

[Gordon Bell, 2009] Gordon Bell, Tony Hey, A. S. (2009). Beyond the data deluge. *Science*, 323:1297–1298.

[Hey et al., 2009] Hey, A., Tansley, S., and Tolle, K. (2009). *The Fourth Paradigm: Data-intensive Scientific Discovery*. Microsoft Research.

[Knab et al., 2006] Knab, P., Pinzger, M., and Bernstein, A. (2006). Predicting defect densities in source code files with decision tree learners. In *MSR*.

[Leek, 2013] Leek, J. (2013). The key word in data science is not data, it is science.

[M. Blei et al., 2003] M. Blei, D., Y. Ng, A., and Jordan, M. (2003). Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022.

[Missier et al., 2010] Missier, P., Soiland-Reyes, S., Owen, S., Tan, W., Nenadic, A., Dunlop, I., Williams, A., Oinn, T., and Goble, C. (2010). Taverna, reloaded. In Gertz, M. and Ludäscher, B., editors, *Scientific and Statistical Database Management*, pages 471–481, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Pavan et al., 2012] Pavan, K. K., Rao, A. A., Rao, A. V. D., and Sridhar, G. R. (2012). Robust seed selection algorithm for k-means type algorithms. *CoRR*, abs/1202.1585.

[Pellin, 2006] Pellin, B. N. (2006). Using classification techniques to determine source code authorship.

[Powers, 2011] Powers, D. M. W. (2011). Evaluation: From precision, recall and f-measure to roc, informedness, markedness & correlation. *Journal of Machine Learning Technologies*.

[Pustejovsky and Stubbs, 2012] Pustejovsky, J. and Stubbs, A. (2012). *Natural Language Annotation for Machine Learning:A Guide to Corpus-Building for Applications*. oreilly.

[Raschka, 2015] Raschka, S. (2015). *Python Machine Learning*. Packt.

[Rennie et al., 2003] Rennie, J. D. M., Shih, L., Teevan, J., and Karger, D. R. (2003). Tackling the poor assumptions of naive bayes text classifiers. In *In Proceedings of the Twentieth International Conference on Machine Learning*, pages 616–623.

[Rule et al., 2018] Rule, A., Tabard, A., and Hollan, J. D. (2018). Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 32:1–32:12, New York, NY, USA. ACM.

[Rumelhart et al., 1986a] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986a). Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, Cambridge, MA, USA.

[Rumelhart et al., 1986b] Rumelhart, D. E., McClelland, J. L., and PDP Research Group, C., editors (1986b). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. MIT Press, Cambridge, MA, USA.

[Stuart and Ord, 1994] Stuart, A. and Ord, J. K. (1994). *Kendall's advanced theory of statistics. Vol.1: Distribution theory*.

[Ugurel et al., 2002] Ugurel, S., Krovetz, R., Giles, C., Pennock, D., Glover, E., and Zha, H. (2002). What's the code? automatic classification of source code archives. In Hand, D., Keim, D., and Ng, R., editors, *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 632–638.

[Zevin and Holzem, 2017] Zevin, S. and Holzem, C. (2017). Machine learning based source code classification using syntax oriented features. *CoRR*, abs/1703.07638.

A

Appendix

A.1 Implementation

This section discusses in brief the implementation details of the python classes and notebooks (path: *src/notebooks/*) developed as a part of this thesis. The implementation language is python and all the classes developed are pure-pythonic and might depend on external libraries. Jupyter notebooks (with python as kernel language) are also developed wherever necessary. More documentation is found within the scripts themselves. df indicates a dataframe in this section.

A.1.1 Feature Generator

Feature Generator includes a set of classes including a Notebook Parser class:CellFeatures. It provides varied set of features (see Section 5.3.1 for definitions) extracted from the notebook not limiting to the textual content of the notebook. The features are then later used to solve the classification problem. The features generated can also be used for other machine learning problems. All the classes take *.ipynb* files as input except for CodeMetrics class which takes *.py* files as input. The classes developed as a part of Feature Generator are explained below.

Notebook Parser

The CellFeatures class The main python class to extract the features from a notebook is implemented in Class.Features.CellFeatures. CellFeatures leverages the json format of the notebooks. For each notebook, multiple features (column) of each cell (row) are extracted. The dataframe is then serialized as a pickle file for further analysis.

classname: CellFeatures

invocation: CellFeatures(path, files, store_path)

path: path to the folder containing the notebooks

files: list of notebooks for which features are to be extracted or provide '[]' for all the files

store_path: path to store the pickle files generated (a .pkl for each notebook is stored separately)

methods: get_code_cell_features()

output: a dataframe containing features from all the cells of the notebooks extracted. Each row in the output dataframe represents a cell in a notebook and each column represents a feature.

Other Features

The PypiPackagesInformationFeatures class PypiPackagesInformationFeatures class uses the import libraries information in a cell as input and generates a new feature based on pypi description of the libraries. It is implemented in Class.Features.PypiPackagesInformationFeatures.

classname: PypiPackagesInformationFeatures

invocation: PypiPackagesInformationFeatures(df, pypi)

df: dataframe of the cell features

pypi: dataframe of the pypi information with columns: library and description

methods: get_pypi_packages_information_features(col, newcol)

col: column name of the dataframe containing library information

newcol: column name of the newly generated feature

output: modified input dataframe (not a copy). Each row in the output dataframe represents a cell in a notebook and the new column contains the respective description of the libraries.

The StyleFeatures class StyleFeatures class extracts style (or) user (notebook creator) related features. It is implemented in Class.Features.StyleFeatures. It is only applicable for metadata format as in GitHub corpus.

classname: StyleFeatures

invocation: StyleFeatures(df_repo, df_readme, df_owner, path, files)

df_repo: dataframe of the repository metadata as in GitHub corpus

df_readme: dataframe of the readme metadata as in GitHub corpus

df_owner: dataframe of the repository owner metadata as in GitHub corpus

path: path to the folder containing the notebooks

files: list of notebooks for which the features are to be extracted or provide '[]' for all the files

methods: get_style_features()

output: a new dataframe containing style/user features (see Section 5.3.1 for definitions) of the list of the notebooks. Each row in the output dataframe represents a notebook.

The CodeMetrics class CodeMetrics class extracts code metrics for each.py file (notebook) using python library radon[1]. It is implemented in Class.Metrics.CodeMetrics.

classname: CodeMetrics

invocation: CodeMetrics(path, files)

path: path to the folder containing the .py files

files: list of .py files (.py files of notebooks) for which the features are to be extracted or provide '[]' for all the files

methods: get_code_metrics()

output: a new dataframe containing code metrics for the list of the notebooks. Each row in the output dataframe represents a notebook.

The NotebookMetrics class NotebookMetrics class extracts metrics like no of code cells, no of markdown cells, no of code tokens, no of markdown tokens per notebook and is implemented in Class.Metrics.NotebookMetrics.

classname: NotebookMetrics

invocation: NotebookMetrics(df)

df: a dataframe containing features from all the cells of the notebooks (as generated by class:CellFeatures)

methods: get_notebook_metrics()

output: a new dataframe containing notebook metrics based on the input df. Each row in the output dataframe represents a notebook.

The CodeCellMetrics class CodeCellMetrics class extracts metrics like no of variables, no of functions, no of lines of code, no of lines of comment per notebook and is implemented in Class.Metrics.CodeCellMetrics.

classname: CodeCellMetrics

invocation: CodeCellMetrics(path, files)

path: path to the folder containing the notebooks

[1]Refer to *https://radon.readthedocs.io/en/latest/* for more information on the metrics

files: list of notebooks for which the features are to be extracted or provide '[]' for all the files

methods: get_popularity_metrics()

output: a new dataframe containing metrics (aggregated over all code cells in a notebook) for the list of the notebooks. Each row in the output dataframe represents a notebook.

The PopularityMetrics class PopularityMetrics class extracts metrics like fork count, star count, watcher count per notebook and is implemented in Class.Metrics.PopularityMetrics. It is only applicable for metadata format as in GitHub corpus.

classname: PopularityMetrics

invocation: PopularityMetrics(df_repo, df_owner, path, files)

df_repo: dataframe of the repository metadata as in GitHub corpus

df_owner: dataframe of the repository owner metadata as in GitHub corpus

path: path to the folder containing the notebooks

files: list of notebooks for which the features are to be extracted or provide '[]' for all the files

methods: get_popularity_metrics()

output: a new dataframe containing popularity metrics for the list of the notebooks. Each row in the output dataframe represents a notebook.

A.1.2 The Preprocessing class

Preprocessing class uses NLTK library to preprocess the text data. The rules are implemented in Class.Preprocessing.Preprocessing for code processing, text processing and import statements processing.

classname: Preprocessing

invocation: Preprocessing(df)

df: dataframe of the cell features

methods:

set_column(self, col, newcol)

col: column name of the dataframe containing text features to be preprocessed

newcol: column name of the newly generated feature ('text_processed')

process(self, newcol, function)

newcol: column name of the newly generated feature

function: function to apply for preprocessing (available functions: code: 'custom_text_preprocessing', import statements: 'import_text_preprocessing', natural language like markdown, comments, raw text: 'text_preprocessing')

output: modified dataframe of input df (not a copy). Each row in the output dataframe represents a cell in a notebook.

A.1.3 The Classifiers class

Classifiers class provides helper functions to set up training data and test data, apply preprocessing methods, tfidf feature representation, and chi2 feature selection. It is implemented in Class.Classifiers.Classifiers.

classname: Classifiers

invocation: Classifiers(df, labels)

df: features in a dataframe format where each row indicates a cell in a notebook (training + validation dataset)

labels: list of classification labels

methods:

apply_conditions_to_dataframe(conditions) *output:* restricted dataframe for machine learning model. For e.g. dataframe containing only cells of cell_type *code*.

test_train_data_set(testdf) *output:* sets test data features

set_lexical_features(features_list) *output:* chooses text based features using FeatureSelector class (see Scripts for more information) for machine learning task and returns train and test data

preprocessing(col) *output:* preprocesses text based features of training + validation and test data and stores it in column *col*

vectorization(tfidf) *output:* vectorizes selected text based features of training + validation and test data using the input tfidf model. returns vectorized training+validation features, test features and trained tfidf

feature_selection(chi2, k, training_labels) *output:* selects k number of features from text based features using the *chi2* method and training_labels. returns selected training+validation features, test features and trained chi2 selector.

set_statistical_features(stat_features, X_train_features, X_test_features) *output:* chooses the specified list of statistical features using FeatureSelector

class (see Scripts for more information) and combines it with the training+validation and test data's vectorized text features and returns the same. The resulting feature set of training+validation and test data is used for training the machine learning model and prediction.

A.1.4 The Clustering class

Clustering class implements LDA, K-means, Agglomerative/Hierarchical clustering and uses scikit-sklearn library for the same. It is implemented in Class.Clustering.Clustering.

classname: Clustering

invocation: Clustering(lda_params, hierarchical_params, results_path, content)

lda_params: (start = start value of range of topics to explore, stop = stop value of range of topics to explore, step = step value of range of topics to explore, top_words = number of words to return per topic, corpus = bag of words of corpus)

hierarchical_params: (max_d = maximum distance between clusters for plotting, full_dendrogram = True if full dendrogram plot is required, truncated_dendrogram = True if truncated dendrogram plot is required, linkage_metric = 'ward', affinity_metric = 'euclidean', h_corpus = vectorized features as array)

results_path: path to store the plot results

content: keyword about the features used (string)

methods:

get_best_clusters_prediction()

output: get_best_clusters_prediction() - number of clusters as predicted by LDA Topic model and Hierarchical clustering

KMeans_model('default' = 'default' or '', Kmeans_num = if not default, use this as seed for n_clusters, vect_corpus = vectorized features as array, vect_feature_names = names of features using tfidf)

output: KMeans_model - features in clusters and kmeans.labels_

A.1.5 Models

All the classifiers are implemented in class.Models.<classifier>. The classifiers are implemented using scikit-learn[2] functions along with GridSearchCV for cross validation. Cross validation folds are given by user input.

[2]scikit-learn *https://scikit-learn.org/stable/*

classname: <classifier>.
Available classifiers are: DT (DecisionTree), GB (Gradient Boosting), KN (KNeighbors), LSVC (Linear Support Vector Classification), LR (LogisticRegression), MLP (Multilayer Perceptron), MNB (Multinomial NaiveBayes), RF (Random Forest), SVC (Support Vector Classification). Use the abbreviation for invocation.

invocation: <classifier>(X_train, y_train, X_test, y_test, indices_train, indices_test, test, labels, results_path, plotname, content, fold)

X_train: training features and validation features

y_train: training + validation labels

X_test: test features

y_test: test labels

indices_train: indices of training and validation dataset

indices_test: indices of test dataset

test: original test features before preprocessing for evaluation

labels: list of classification labels

results_path: path to store the results

plotname: name of the plot to be generated (string)

content: keyword about the features used (string)

fold: cross validation fold (int) or PredefinedSplit or KFold or StratifiedKFold split for cross validation

methods: run()

output: The classifier classes along with the prediction output also return evaluation scores like f1score, accuracy, classification report, confusion matrix and probabilities (if available). Predicted classes against their features are stored in result .pkl. The classes also return the trained model that can be readily used on new preprocessed inputs.

A.1.6 Utils

The GenerateLabelFiles and GenerateExperimentLabelFiles class GenerateLabelFiles and GenerateExperimentLabelFiles classes creates annotation template for main annotation and annotation experiment respectively to get classification labels and is implemented in Class.Utils.GenerateLabelFiles and Class.Utils.GenerateExperimentLabelFiles.

classname: GenerateLabelFiles/GenerateExperimentLabelFiles

invocation: GenerateLabelFiles(pickle_files, pickle_path, labels_path)/ GenerateExperimentLabelFiles(pickle_files, pickle_path, labels_path

pickle_files: list of the files for which annotation template files are to be generated

pickle_path: path to the folder containing the pickles (features of the notebook obtained using CellFeatures)

labels_path: path to the folder where output files are to be stored

methods: generate_label_files()

output: .csv/.xlsm file for each of the notebooks are stored in labels_path

The KaggleNotebooks class KaggleNotebooks class finds and downloads Kaggle notebooks for a given search keyword. It is implemented in Class.Utils.KaggleNotebooks.

classname: KaggleNotebooks

invocation: KaggleNotebooks(path, search_keyword)

path: path to store the downloaded Kaggle notebooks

search_keyword: keyword to search for Kaggle notebooks using Kaggle API

methods: get_kaggle_notebooks()

output: all the notebooks found for a given search keyword are stored in the path specified and respective metadata information for all the notebooks are returned as a dataframe

The WordCloudView class WordCloudView creates a word cloud from words and frequencies. It is implemented in Class.Utils.WordCloudView and uses external library WordCloud.

classname: WordCloudView

invocation: WordCloudView(c, plotname, path)

c: words and frequencies

plotname: name of the WordCloud image

path: path to store the generated WordCloud image

methods: plot()

output: WordCloud image for words and frequencies input stored in path specified

The CustomPlot class CustomPlot creates a custom plot from the data based on input parameters. It is implemented in Class.CustomPlot.CustomPlot. It is used for Exploratory Data Analysis.

classname: CustomPlot

invocation: CustomPlot(seaborn_palette)

methods: plot_seaborn_category(dataframe_data, column_to_plot, top_count_to_plot, start, plotname, path_to_store_plot, fig, axes)

start: start value for top values to be plotted. For e.g plot top first 20 values (start = 0) or plot top 20 values from second top (start = 1)

output: custom plot generated and stored in path specified

The PypiPackagesInformation class PypiPackagesInformation retrieves language, summary and description of all the libraries available in PyPi[3] and is implemented in Class.Utils.PypiPackagesInformation. Output from PypiPackagesInformation is used by class:PypiPackagesInformationFeatures.

classname: PypiPackagesInformation

invocation: PypiPackagesInformation()

methods: get_pypi_packages_information()

output: dataframe containing libraries information, its language, summary and description from PyPi

The GetClassifierReport class GetClassifierReport class creates output format for annotation experiment to get classification labels and is implemented in Class.Utils.GetClassifierReport.

classname: GetClassifierReport

invocation: GetClassifierReport(results_path, plotname, accuracy, f1score, classification_report, confusion_matrix, confusion_matrix_as_string, result_df, classification_labels)

result_df: containg labels predicted, probabilities (if available) and corresponding text feature in a dataframe

methods: get_reports()

output: all the evaluation metrics are used to create a report and is stored in the result_path along with the test results

The PandasUtils class PandasUtils implements helper functions to load multiple pickle files or csv files into a single dataframe and is implemented in Class.Utils.PandasUtils.

classname: PandasUtils

invocation: PandasUtils()

[3]PyPi *https://pypi.org/*

methods:

get_pickle_files_in_dataframe(pickle_path, pickle_files)

pickle_files: list of the pickle files to be combined

pickle_path: path to the folder containing the pickles

get_csv_files_in_dataframe(csv_path, csv_files)

csv_files: list of the csv files to be combined

csv_path: path to the folder containing the csvs

output: a dataframe with the all pickles/csvs combined

A.1.7 Notebooks

I implemented several notebooks as a part of the thesis in order to implement the tasks like data_preparation, classification, annoatation etc. In this section, the purpose of the notebooks are explained in brief. More comments, explanations and parameter setups are found in the notebooks (for the restriction of space). The numeric in front of the filename indicate the order in which notebooks shall be executed (specially for the data preparation process).

1. 1_prep_choose_valid_filenames_from_pickles: This notebook is designed to extract notebooks that are valid and json readable from the corpus.

2. 2_prep_selection_of_test_train_simulation_dataset.ipynb: This notebook implements functions required for selecting data set (training, test) by random sampling.

3. 3_prep_generate_classification_label_files.ipynb: This notebook implements functions to generate annotation template files (main annotation and annotation experiment by other experts) for recording classification labels by data science experts

4. 4_prep_kaggle_notebook_corpus_preparation.ipynb: This notebook retrieves public kaggle kernels using KaggleAPI and prepares Kaggle Notebook corpus for the experiment.

5. analysis_clustering.ipynb: This notebooks is generated to analyse the performance of unsupervised methods like Topic modelling, Agglomerative clustering and KMeans Clustering on the data.

6. analysis_code_statistical_features.ipynb: This notebook performs exploratory data analysis of the statistical features of the dataset.

7. analysis_import_libraries_composition_in_the_datasets.ipynb: This notebook is implemented to analyse the usage of external libraries (top libraries in dataset, number of libraries imported per notebook etc.)

8. analysis_language_composition_in_the_datasets.ipynb: This notebook analyses the language composition in the dataset.

9. analysis_notebook_style_features.ipynb: This notebook is implemented to analyse if there is any relation between popularity features and other features in the notebook dataset.

10. data_preparation.ipynb: This notebook is implemented to prepare the data (from the corpus) for the classification experiment. Various features generated using the classes available in Feature Generator (see Section A.1.1) are also explained.

11. eda_features.ipynb: This notebook is implemented to perform exploratory analysis of the features generated.

12. inter_annotator_agreement.ipynb: This notebook analyses the inter-annotator agreement between the main annotation and other annotations (gathered through experiment) using the percentage of agreement and cohen-kappa metric.

13. model_question_3.ipynb: This notebook is implemented to generate *'text'* feature based on only import statements in the notebooks in order to answer the research question 3.

14. modelling_multi_label_classification.ipynb: This notebook is implemented to perform classification of the dataset using Singlelabel Multiclass classification.

15. modelling_single_label_classification.ipynb: This notebook is implemented to perform classification of the dataset using Multilabel Multiclass classification.

16. rdf_ontology_annotator.ipynb: This notebooks uses ***'no'*** ontology to annotate notebooks based on their classification labels and produces an annotated dataset serialized in RDF[4].

17. theory_visualize_datascience_in_theory_and_practice.ipynb: This notebook analyses various steps currently suggested in a data science pipeline by academia, industry, and individual data science experts.

A.2 Data Science Ontology for Notebooks ('no' ontology)

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://www.semanticweb.org/ramas/ontologies/2019/0/no"
xml:base="http://www.semanticweb.org/ramas/ontologies/2019/0/no"
xmlns:no="http://www.semanticweb.org/ramas/ontologies/2019/0/no#"
xmlns:owl="http://www.w3.org/2002/07/owl#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:xml="http://www.w3.org/XML/1998/namespace"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
```

[4]Resource Description Framework *https://en.wikipedia.org/wiki/Resource_Description_Framework*

```
<owl:Ontology rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no">
<owl:versionIRI rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no/1.0.0"/>
</owl:Ontology>

    <!-
//////////////////////////////////////////////////////////////////////////////////
//
// Object Properties //
//////////////////////////////////////////////////////////////////////////////////
->

    <!- http://www.semanticweb.org/ramas/ontologies/2019/0/no#hasDataScienceActivity ->

    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#hasDataScienceActivity">
<rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topObjectProperty"/>
<owl:inverseOf rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#isDataScienceActivityIn"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Code"/>
<rdfs:range rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#CommentOnly"/>
<rdfs:range rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataExploration"/>
<rdfs:range rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataPreprocessing"/>
<rdfs:range rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Evaluation"/>
<rdfs:range rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#HelperFunctions"/>
<rdfs:range rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#LoadData"/>
<rdfs:range rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Modelling"/>
<rdfs:range rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Prediction"/>
<rdfs:range rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#ResultVisualization"/>
<rdfs:range rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#SaveResults"/>
</owl:ObjectProperty>

    <!- http://www.semanticweb.org/ramas/ontologies/2019/0/no#hasNotebookCell ->

    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#hasNotebookCell">
<rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topObjectProperty"/>
<owl:inverseOf rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#isNotebookCellOf"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Notebook"/>
<rdfs:range rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#NotebookCell"/>
</owl:ObjectProperty>

    <!- http://www.semanticweb.org/ramas/ontologies/2019/0/no#isDataScienceActivityIn ->

    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#isDataScienceActivityIn">
<rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topObjectProperty"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#CommentOnly"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataExploration"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataPreprocessing"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Evaluation"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#HelperFunctions"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#LoadData"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Modelling"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Prediction"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#ResultVisualization"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#SaveResults"/>
<rdfs:range rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Code"/>
</owl:ObjectProperty>

    <!- http://www.semanticweb.org/ramas/ontologies/2019/0/no#isNotebookCellOf ->

    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#isNotebookCellOf">
<rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topObjectProperty"/>
```

<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#NotebookCell"/>
<rdfs:range rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Notebook"/>
</owl:ObjectProperty>

<!–
///
//
// Data properties
//
///
–>

<!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#cell_number –>

<owl:DatatypeProperty rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#cell_number">
<rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topDataProperty"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#NotebookCell"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
</owl:DatatypeProperty>

<!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#file_extension –>

<owl:DatatypeProperty rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#file_extension">
<rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topDataProperty"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Notebook"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#name –>

<owl:DatatypeProperty rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#name">
<rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topDataProperty"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Notebook"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#owner –>

<owl:DatatypeProperty rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#owner">
<rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topDataProperty"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Notebook"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#Name"/>
</owl:DatatypeProperty>

<!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#programming_language –>

<owl:DatatypeProperty rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#programming_language">
<rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topDataProperty"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Notebook"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#published_in_platform –>

<owl:DatatypeProperty rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#published_in_platform">
<rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topDataProperty"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Notebook"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>

```
</owl:DatatypeProperty>

    <!-- http://www.semanticweb.org/ramas/ontologies/2019/0/no#type -->

    <owl:DatatypeProperty rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#type">
<rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topDataProperty"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Code"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Markdown"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#RawNB"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

    <!-- http://www.semanticweb.org/ramas/ontologies/2019/0/no#url -->

    <owl:DatatypeProperty rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#url">
<rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topDataProperty"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Notebook"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#anyURI"/>
</owl:DatatypeProperty>

    <!-- http://www.semanticweb.org/ramas/ontologies/2019/0/no#workflow_system -->

    <owl:DatatypeProperty rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#workflow_system">
<rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topDataProperty"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Notebook"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

    <!-- http://www.semanticweb.org/ramas/ontologies/2019/0/no#workflow_type -->

    <owl:DatatypeProperty rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#workflow_type">
<rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topDataProperty"/>
<rdfs:domain rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Notebook"/>
<rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

    <!--
///////////////////////////////////////////////////////////////////////////////////
//
// Classes
//
///////////////////////////////////////////////////////////////////////////////////
-->

    <!-- http://www.semanticweb.org/ramas/ontologies/2019/0/no#Code -->

    <owl:Class rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Code">
<rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#NotebookCell"/>
</owl:Class>

    <!-- http://www.semanticweb.org/ramas/ontologies/2019/0/no#CommentOnly -->

    <owl:Class rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#CommentOnly">
<rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataScienceWorkFlow"/>
</owl:Class>

    <!-- http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataExploration -->
```

<owl:Class rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataExploration">
<rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataScienceWorkFlow"/>
</owl:Class>

<!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataPreprocessing –>

<owl:Class rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataPreprocessing">
<rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataScienceWorkFlow"/>
</owl:Class>

<!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataScienceWorkFlow –>

<owl:Class rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataScienceWorkFlow"/>

<!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#Evaluation –>

<owl:Class rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Evaluation">
<rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataScienceWorkFlow"/>
</owl:Class>

<!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#HelperFunctions –>

<owl:Class rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#HelperFunctions">
<rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataScienceWorkFlow"/>
</owl:Class>

<!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#LoadData –>

<owl:Class rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#LoadData">
<rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataScienceWorkFlow"/>
</owl:Class>

<!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#Markdown –>

<owl:Class rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Markdown">
<rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#NotebookCell"/>
</owl:Class>

<!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#Modelling –>

<owl:Class rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Modelling">
<rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataScienceWorkFlow"/>
</owl:Class>

<!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#Notebook –>

<owl:Class rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Notebook"/>

<!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#NotebookCell –>

<owl:Class rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#NotebookCell">
<rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Notebook"/>
</owl:Class>

<!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#Prediction –>

<owl:Class rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#Prediction">
<rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataScienceWorkFlow"/>

```
</owl:Class>

    <!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#RawNB –>

    <owl:Class rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#RawNB">
<rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#NotebookCell"/>
</owl:Class>

    <!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#ResultVisualization –>

    <owl:Class rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#ResultVisualization">
<rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataScienceWorkFlow"/>
</owl:Class>

    <!– http://www.semanticweb.org/ramas/ontologies/2019/0/no#SaveResults –>

    <owl:Class rdf:about="http://www.semanticweb.org/ramas/ontologies/2019/0/no#SaveResults">
<rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ramas/ontologies/2019/0/no#DataScienceWorkFlow"/>
</owl:Class>
</rdf:RDF>

    <!– Generated by the OWL API (version 4.5.7.2018-12-02T02:23:35Z) https://github.com/owlcs/owlapi –>
```


A.3 Expert Annotation Experiment

The main goal of this master thesis is to design and implement a method that automatically classifies different parts of a Data Science notebook, so as to label the steps of the Data Science process that are present in the notebook. In order to train and evaluate the method, we need to curate a ground truth. Therefore, your input as an expert labeller is very much appreciated.

You will be given a set of notebooks to be labelled, a set of possible labels and your task is to indicate for each cell of the notebook one or more relevant label(s), that describe the purpose of the code appearing in the cell. We would also like you to indicate how confident you feel about your answers. Below, you will find detailed information about the meaning of the labels and things to take into account.

In this annotation task, you will need to consider the following files for each notebook to be annotated:

- *notebook_ name.ipynb*: the Python notebook to be analysed and labelled.
- *notebook_ name.ipynb.xlsm*: a macro-enabled excel worksheet, where you should annotate the notebook.

A.3.1 Instructions

Please read the instructions before proceeding with the labelling task.

How to annotate one notebook

- Open the Python notebook and the annotation file in any compatible software (e.g., use Jupyter for the notebook and Excel for the annotation file).
- Read the set of possible labels and their meaning (see also Section A.3.1).
- For each cell in the notebook, please read the content and consider the line(s) of code, plots, visualizations and comments. Please ignore markdown.
- Decide the purpose of the cell, and assign one or more of the available labels. In the annotation file (*notebook_ name.ipynb.xlsm*), please mark with *1* the label(s) you would like to assign to the cell.
- Also, determine the primary data science process performed in the cell. Record the observation by choosing the appropriate label in the primary_label column in the annotation file.
- If, for a given cell, you do not agree with any of the given labels, please indicate the label that you think is relevant in the *other_ label* column.
- You have the option of adding some notes in the *notes* column, if you would like to enter any observation about the cell (e. g. if it was hard to decide the label or anything you think it is important to be mentioned). You do not need to enter notes for every single cell.
- Once you have completed the labelling of all cells, please indicate in the last two rows of annotation file the following information:
    1. confidence percentage: how confident you felt when providing the labels (with a value between 0 (not confident at all) and 100 (very confident)).
    2. notebook score: give a general score for the notebook, based on the clarity and understandability of the code (with a value between 0 (not clear at all) and 100 (very clear)).

General tips

· The annotation should be done based on the available implementation in the notebook. Please do not execute the code of the notebook to decide the label.

· Note that sometimes, it is useful to go through the whole notebook to be aware of the context of the cells.

Classification Labels

We have defined a set of 9 labels: 3 of them are general labels that provide some meta-information (see Section A.3.1), and 7 labels correspond to the main steps in the data science process (see Section A.3.1).

General Labels

load_data This label indicates that the cell is just loading data into the Jupyter notebook environment. The code can be for loading one or multiple data sets of any type (e.g. .csv, .pkl, .jpg, .png, .hdf5).

helper_functions This label should be used when the cell contains code that supports the code in the notebooks. These helper functions can be *import statements* or other piece of code that is not directly related to the data science activity at hand and rather are useful functions in scripting. For example, built-in Jupyter notebook magic commands. A concrete example would be: *%matplotlib inline*, which sets the inline backend so that the output plots are displayed directly below the code cell that executes it. Some other examples include `import pandas as pd`, `from IPython.display import Audio`, and `%pprint`.

comment_only If the cell contains only commented text, you should use this label. This label should not be used for markdown; please remember that you should ignore markdown.

Labels for Each of the Data Science Steps

data_preprocessing Data preprocessing includes tasks such as cleaning, instance selection, data normalization, data transformation, feature extraction and feature selection. Data preprocessing ensures that the data does not contain irrelevant, redundant and inconsistent data. [5]

data_exploration Data exploration[6] or exploratory data analysis is an approach to initial data analysis, where a data scientist inspects the content of a dataset in order to understand the nature and characteristics of the data. This step, that often contributes to the identification of patterns in the data, is helpful to identify research hypotheses and decide on the modelling. Data exploration often involves visual exploration of the data.

modelling Modelling is the process of applying (or fitting) statistical models and algorithms to the data in order to "perform a specific task without explicit restrictions, relying on models and inferences instead"[7]. In simple terms, training a classifier or a neural network to learn from the data is modelling.

prediction Prediction is an important step as many of the data science tasks are predictive modelling tasks. This label refers to the generation of an outcome that was previously unknown, by applying the model built to new data.

[5] Data preprocessing *https://en.wikipedia.org/wiki/Data_pre-processing*
[6] Data exploration *https://en.wikipedia.org/wiki/Data_exploration*
[7] Machine learning Models *https://en.wikipedia.org/wiki/Machine_learning#Models*

evaluation Evaluation refers to the process of evaluating the model using evaluation metrics like accuracy, or F1 score. In the context of model selection, the evaluation step can be used to compare different models. valuation is also done to compare different models [8].

result_visualization Result visualization is the graphical representation of results (via elements like plots, tables and other graphs). Please note that we differentiate between data exploration and result visualization. The purpose of the former is to better understand the data, while the latter focuses on visualizing e. g. a tested hypothesis, or a performance comparison.

save_results This label should be used when the cell is primarily creating a persistent copy of some results (e. g. serializing the content of a variable into a CSV file).

Thank you very much for your help.

A.4 Unsupervised classification results

A.4.1 LDA

Figure A.1 shows the visualization of topic modelling of the dataset taking into account all of the code-markdown-raw data. Figure A.2 and Figure A.3 shows the topic modelling of the dataset taking into account code data and import statements data respectively.

[8] Evaluating Machine Learning Models *https://www.oreilly.com/data/free/evaluating-machine-learning-models.csp* for more information

```
Best number of topics:  8
Running the best model with 8...
Top 10 words per topic...
vocabulary:  ['##', 'player', 'number', 'data', 'age', 'image', 'based', 'would', 'cell', '#####']
vocabulary:  ['print', 'data', 'lab', 'df', 'labels', 'axis', 'input', 'train.csv', '..', 'drop']
vocabulary:  ['data', 'integer', '###', 'use', 'taz', '##', 'population', 'function', 'tweets', 'get']
vocabulary:  ['jeopardy', 'model', 'number', 'dataset', '###', 'one', 'also', 'https', 'using', 'question']
vocabulary:  ['import', 'numpy', 'np', 'pandas', 'caravan', 'pd', 'matplotlib', 'output', 'dependency', 'return']
vocabulary:  ['print', 'ax', 'def', 'len', 'return', 'x_train', 'y_train', 'model', 'range', 'y_test']
vocabulary:  ['datos', 'con', 'labels', 'return', 'def', 'bins', 'range', 'pd.read_csv', 'pairs', 'código']
vocabulary:  ['de', 'que', 'https', 'en', 'para', 'es', 'pca', 'la', 'análisis', 'un']
Perplexity:  -8.614875242713468
Coherence Score:  0.4362750225569397
Best number of clusters using lda model:  8
```

Figure A.1: Topic Modelling using LDA (code-markdown-raw)

```
Best number of topics:  6
Running the best model with 6...
Top 10 words per topic...
vocabulary:  ['data', 'return', 'def', 'axis', 'print', 'true', 'len', 'shape', 'charges', 'mask']
vocabulary:  ['import', 'numpy', 'np', 'data', 'pandas', 'pd', 'matplotlib', 'plt', 'matplotlib.pyplot', 'inline']
vocabulary:  ['return', 'def', '', 'row', 'df', 'alpha', 'range', 'list', 'bins', 'len']
vocabulary:  ['print', 'y_train', 'x_train', 'y_test', 'len', 'x_test', 'error', 'true', 'data', 'batch_size']
vocabulary:  ['def', 'return', 'color', 'axis', 'df', 'true', 'model.add', 'alpha', 'year', 'y_pred']
vocabulary:  ['ax', '..', 'input', 'pd.read_csv', 'false', 'print', 'fontsize', 'range', 'cmap', '.set_title']
Perplexity:  -7.8009350807337094
Coherence Score:  0.43657034445268045
Best number of clusters using lda model:  6
```

Figure A.2: Topic Modelling using LDA (code)

A.4.2 Agglomerative Clustering

Figure A.4 shows the dendrogram visualization (n_components=3) of agglomerative clustering of the dataset taking into account all of the code-markdown-rawnb data. Figure A.5 and Figure A.6 shows the agglomerative clustering of the dataset taking into account the code data and import statements data respectively.

A.4.3 KMeans Clustering

Figure A.7 shows the visualization of K-means clustering of the dataset taking into account all of the code-markdown-raw data. Figure A.8 and Figure A.9 shows the K-means clustering of the dataset taking into account the code data and import statements data respectively.

```
Best number of topics:  2
Running the best model with 2...
Top 10 words per topic...
vocabulary:  ['', 'tensorflow', 'cv2', 'datetime', 'keras', 'scikit-learn', 'train_test_split', 'layers', 'problem_unittests', 'tyssue
vocabulary:  ['matplotlib', 'numpy', 'pandas', 'seaborn', 'scikit-learn', 'os', 'pd', 'data', 'pyplot', 'np']
Perplexity:  -3.5192379925729327
Coherence Score:  0.5447735296344758
Best number of clusters using lda model:  2
```

Figure A.3: Topic Modelling using LDA (import statements)

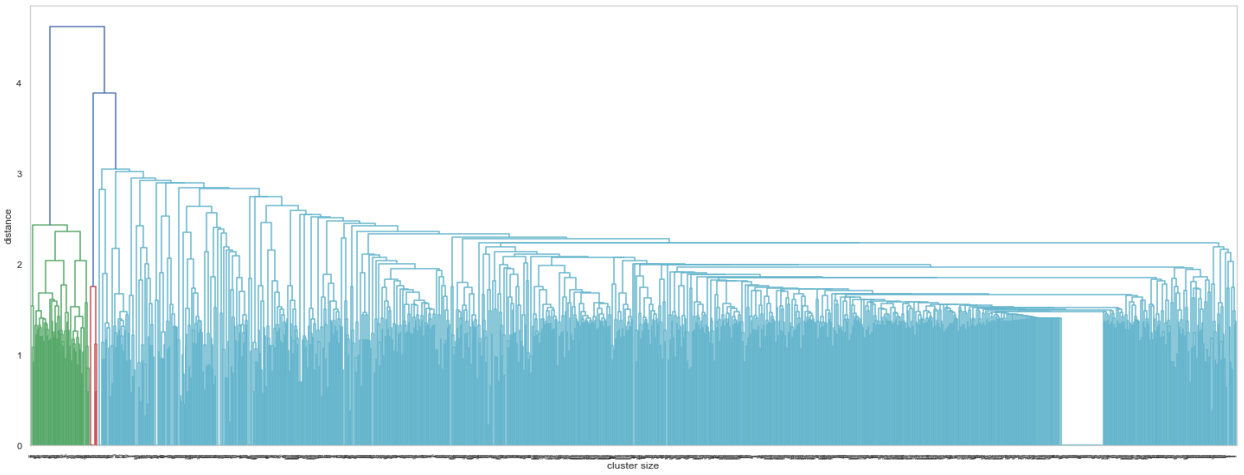

Figure A.4: Agglomerative Clustering - Full view (code-markdown-raw)

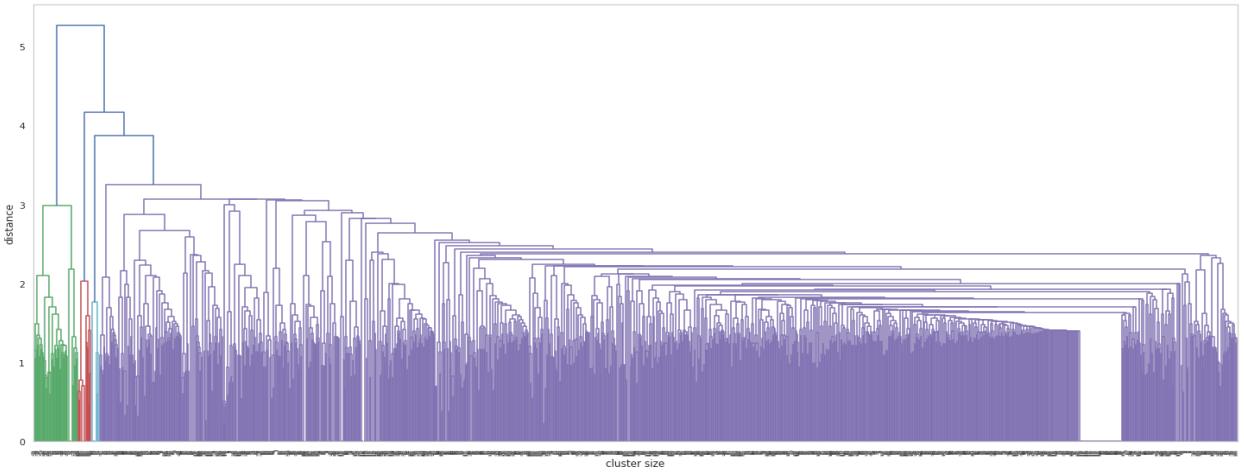

Figure A.5: Agglomerative Clustering - Full view (code)

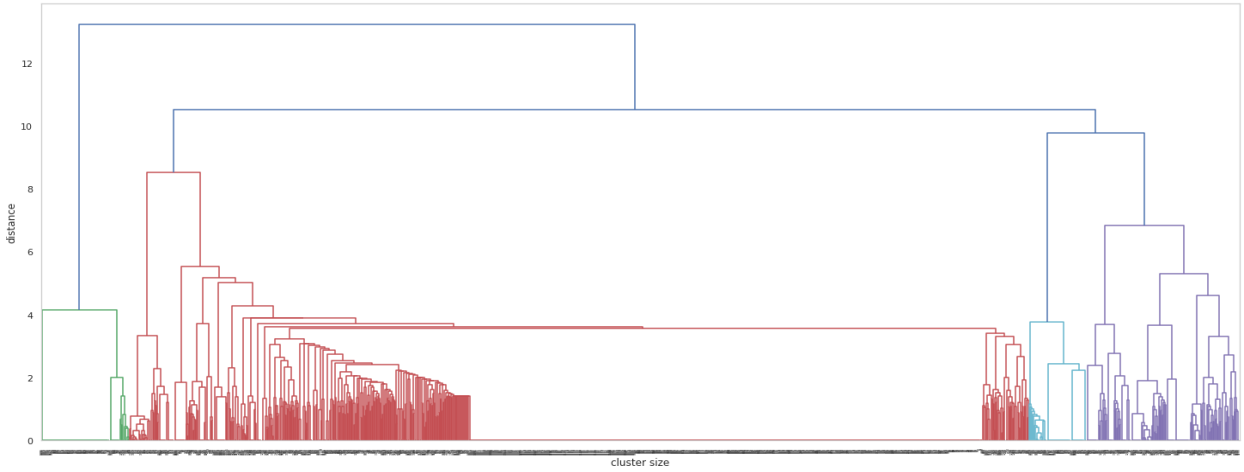

Figure A.6: Agglomerative Clustering - Full view (import)

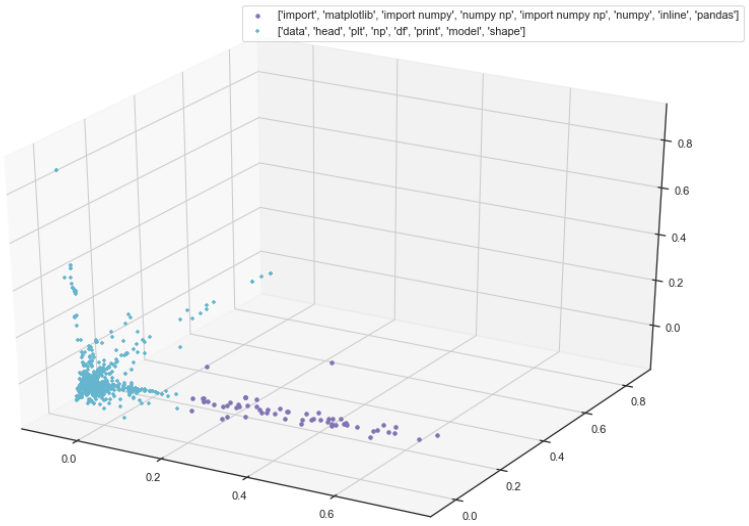

Figure A.7: Kmeans visualized using PCA (code-markdown-raw)

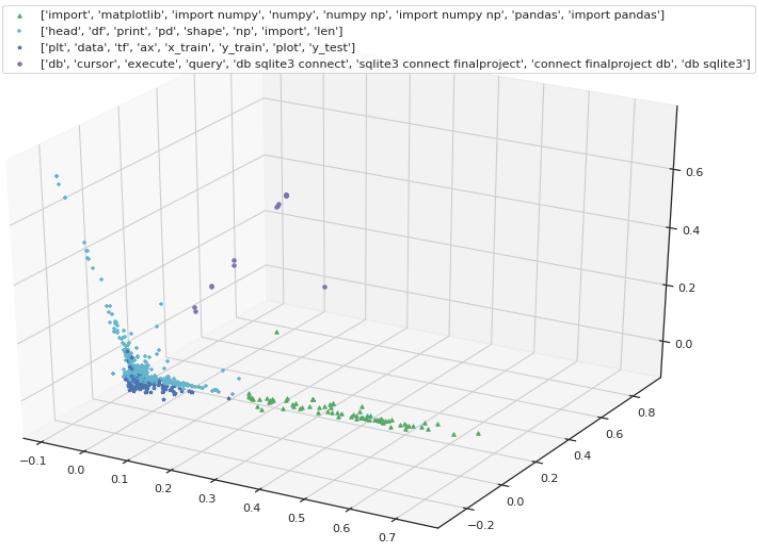

Figure A.8: Kmeans visualized using PCA (code)

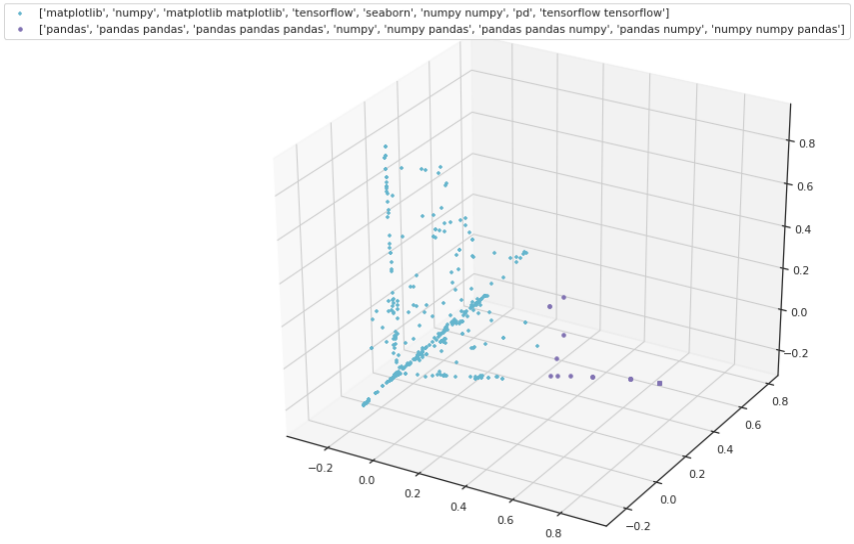

Figure A.9: Kmeans visualized using PCA (import)

A.5 Methods, Tools and Techniques

In this section, I briefly summarize the popular classifiers that are used in the classification task: both parametric[9] (Support Vector Machine, Linear Support Vector Machine, Multinomial Naive Bayes) and non-parametric (K-Nearest Neighbor, Logistic Regression, Multilayer Perceptron, Decision Tree, Random Forest, Gradient Boosting) classifiers applied to the task and the two types of classification task: Singlelabel Multiclass Classification and Multilabel Multiclass Classification performed on the dataset. This section also discusses in brief the two important strategies of multiclass classification: OnevsOne (OvO) and OnevsRest (OvR).

A.5.1 Unsupervised Techniques: Topic Models and Clustering

Latent Dirichlet Allocation (LDA)

Latent Dirichlet Allocation is a generative statistical[10] topic modelling technique "that allows sets of observations to be explained by unobserved groups that explain why some parts of the data are similar." LDA views each document as a mixture of a small number of topics. Each topic represents a set of words and is latent (hidden). LDA aims to use this latent information to predict the number of latent topics and assign topics to each document (see [M. Blei et al., 2003] for the mathematical foundation behind LDA). The important assumption LDA makes is that the prior topic distribution is a symmetric Dirichlet. The required top-document and topic-word distribution for LDA is then modelled using probability density function of Dirichlet distribution[11]:

$$f\left(x_1, \ldots, x_K;\ \alpha_1, \ldots, \alpha_K\right) = \frac{1}{\mathrm{B}(\boldsymbol{\alpha})} \prod_{i=1}^{K} x_i^{\alpha_i - 1}$$

It is to be noted that the order of the words are ignored in LDA as Bag-of-words[12] technique is used to represent documents. Figure A.10 shows the plate diagram for a LDA illustrating the relationship among documents, topics, and words [Bonaccorso, 2018].

In the plate diagram, α is the Dirichlet parameter for the topic-document distribution, while γ is the Dirichlet parameter the topic-word distribution. θ is the topic distribution for a specific document, while β is the topic distribution for a specific word.

K-Means

Clustering is an unsupervised machine learning technique that allows us to find groups of similar instances where the instances are more related to each other within the group

[9] Parametric models make an assumption about the underlying distribution of the data whereas non-parametric models do not. *http://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning/*

[10] *https://en.wikipedia.org/wiki/Generative_model*

[11] *https://en.wikipedia.org/wiki/Dirichlet_distribution*

[12] *https://en.wikipedia.org/wiki/Bag-of-words_model*

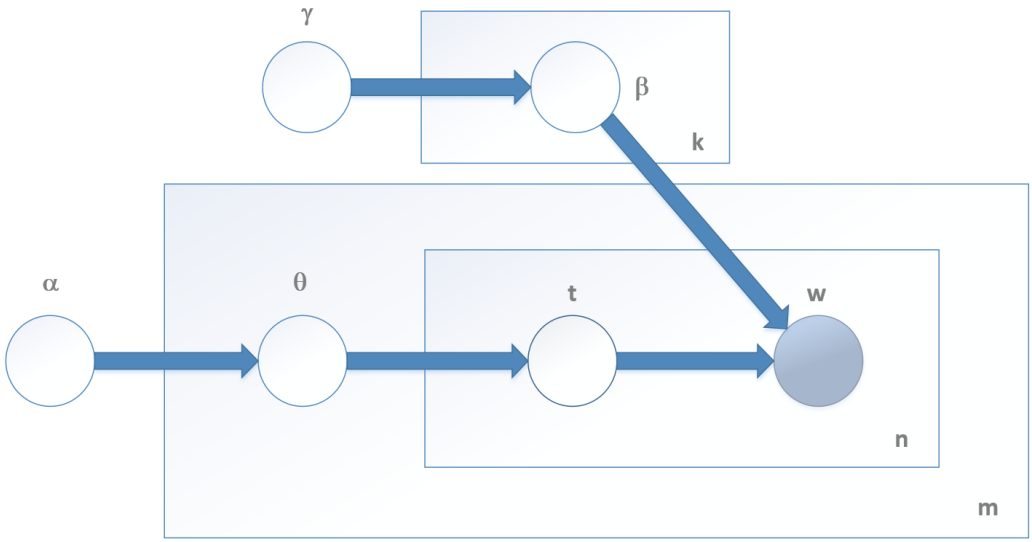

Figure A.10: Plate diagram for a LDA

than to instances in other groups [Raschka, 2015]. One of the most popular clustering algorithms, K-means, aims to partition n instances into k (set as hyperparameter) clusters in which each instance belongs to the cluster with the nearest mean/centroid. As a result of the clustering, the data is partitioned into Voronoi cells. Figure A.11 shows the results of simple k-means clustering.

Steps involved in k-means are:

1. Initialize k cluster centers randomly.

2. Assign step: asssign instances to the closest cluster center. One of the commonly used distance metric is Euclidean distance[13].

3. Update step: update cluster centers as mean of assigned instances of the new clusters.

4. Repeat step 2 and step 3 until convergence, that is, until the assignments no longer change.

There are several techniques to choose k value (e.g. elbow method)[14]. K-means algorithm does not guarantee global optimum as the results depend on initial clusters. Several improvements on K-means have been proposed, the most popular being K-means++ which is an algorithm for choosing initial cluster centers to K-means [Pavan et al., 2012].

[13] *https://en.wikipedia.org/wiki/Euclidean_distance*

[14] Please refer to *https://en.wikipedia.org/wiki/Determining_the_number_of_clusters_in_a_data_set* for more information

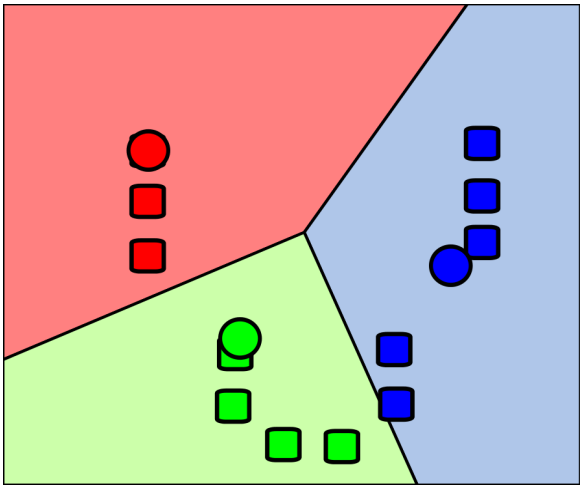

Figure A.11: Results of a simple k-means clustering showing voronoi partitions

A.5.2 Supervised Techniques: Classifiers

Support Vector Machine (SVM)

Support Vector Machine is a popular classifier whose optimization objective is to maximize the distance (functional margin) between separating hyperplane/s and its support vectors. Support vectors are the closest points that identify the separating hyperplane/s. Larger the functional margin, smaller the generalization error of the error. Figure A.12[15] illustrates the hyperplane, support vectors and functional margin of a SVM classifier.

Linear and Non-linear SVM Linear SVM are used in the problems that have linearly separable classes like the one depicted in Figure A.12. Most of the real world problems usually are non-linear classification problem. These problems contains classes that are not separable by linear hyperplane/decision boundary (see Figure A.13). A variant of SVM, non-linear SVM kernels are used to solve non-linear classification problems. The basic idea is to transform the original data into high dimensional feature space using a mapping function φ and then train a linear SVM to classify the data. For example, a linear SVM can be applied to a two-dimensional data (x_1, x_2) after transforming it into a three-dimensional feature space: $\varphi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$ [Raschka, 2015]. The same transformation is applied to new, unseen data before classifying them using SVM. Figure A.14 illustrates the idea of transforming features into a higher dimensional space.

Transformation of orginial features into a higher dimensionall features space requires computing dot-products and is often very expensive. But this can be avoided by applying the kernel-trick: 'For all $\mathbf{x}$ and $\mathbf{x}^{'}$ in the input space $\mathcal{X}$, certain functions $k(\mathbf{x}, \mathbf{x}^{'})$ can

[15] *https://en.wikipedia.org/wiki/Support-vector_machine*

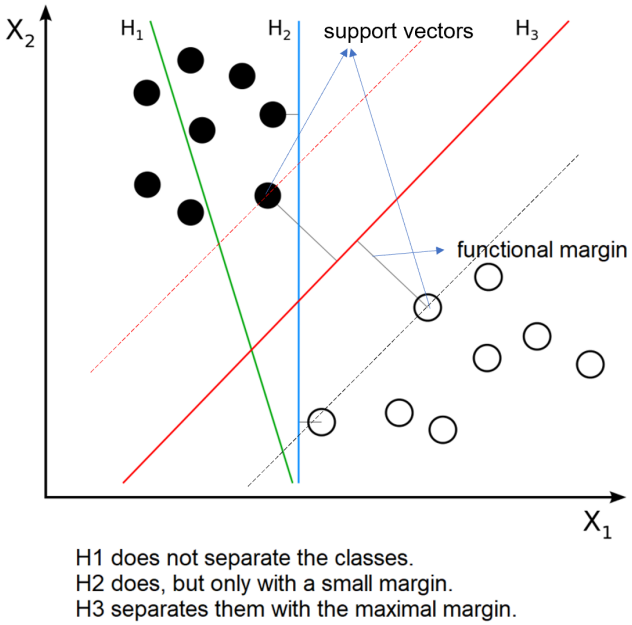

Figure A.12: SVM Classifier

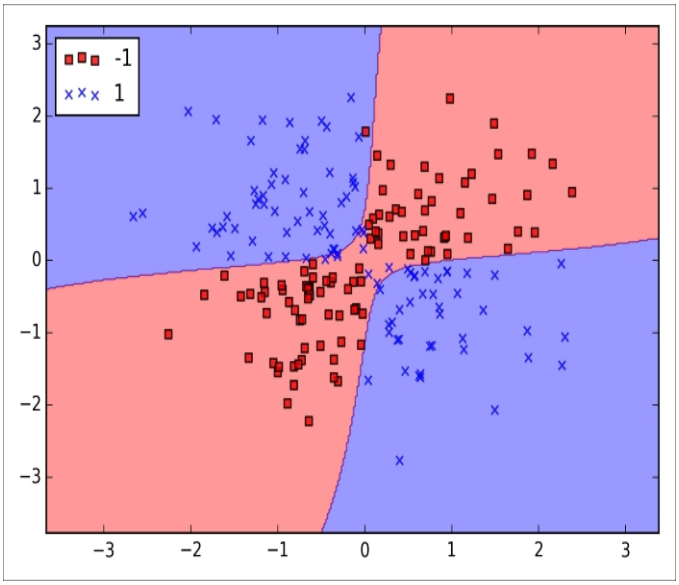

Figure A.13: A simple example of a non-linearly separable data

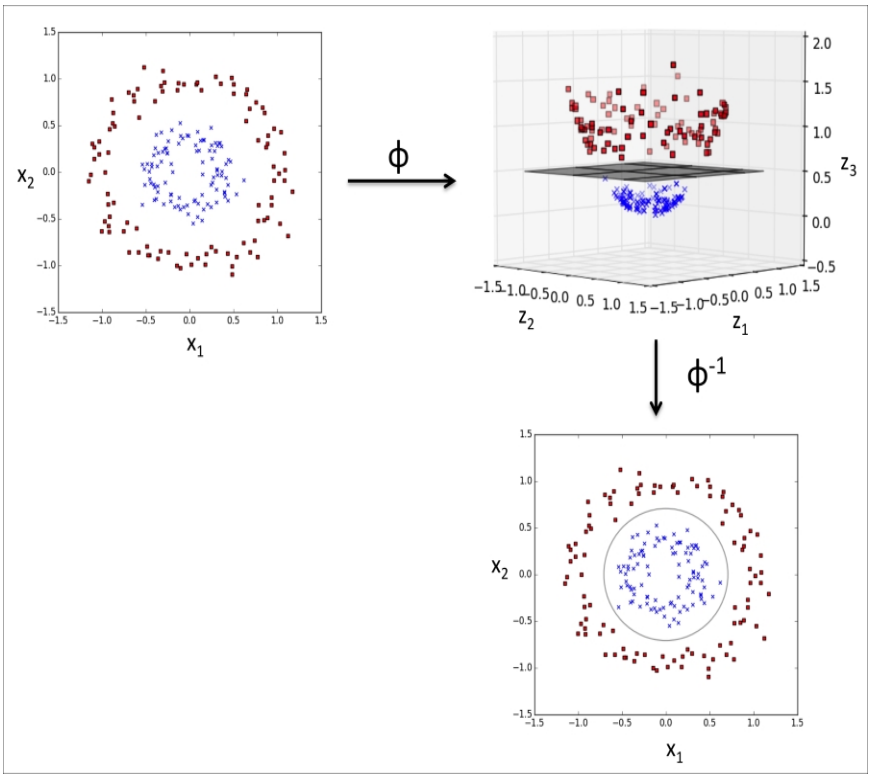

Figure A.14: Transformation of feature space

be expressed as an inner product in another space $\mathcal{V}$, i.e, $k(\mathbf{x}, \mathbf{x}') = \langle\varphi(\mathbf{x}), \varphi(\mathbf{x}')\rangle_{\mathcal{V}}$'[16]. Kernels operate in an high-dimensional implicit features space and thus, by replacing the dot-product by kernel function: , a computationally expernsive explicit mapping is avoided. One of the widely used kernel fucntion is gaussian kernel or Radial Basis Function (rbf) kernel which is denote by $k(\mathbf{x}, \mathbf{x}') = \exp(-\gamma\|\mathbf{x} - \mathbf{x}'\|^2)$[17].

Multinomial Naive Bayes (Multinomial NB)

Multinomial Naive Bayes comes from a family of Naive Bayes classifiers. Naive Bayes classifiers[18] are probabilistic classifiers on based Bayes' theorem [Stuart and Ord, 1994] and have a strong assumption that the features are independent of each other.

Bayes' theorem is mathematically stated as:

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

where A and B are events and $P(B) \neq 0$.

$P(A \mid B)$ is a conditional probability: the likelihood of event A occurring given that B is true.

$P(B \mid A)$ is a conditional probability: the likelihood of event B occurring given that A is true.

$P(A)P(A)$ and $P(B)P(B)$ are the probabilities of observing A and B independently of each other which is known as the marginal probability[19].

Let's consider a dataset with n instances and m features. Every feature vector will be represented as: $x_i = (x_i(1), ...x_i(m))$ and the target vector Y with P classes will be $Y = (y_1, ...y_n)$ where $y_i(0, 1, 2....P - 1)$ where each y_i belongs to one of P classes.

Under conditional independence, Bayes' theorem can be written as:

$$P(y_i \mid x_i(1), ...x_i(m) = \alpha P(y_i) \textstyle\prod_j P(x_i{}^{(j)} \mid y_i)$$

where marginal Apriori probability $P(y_i)$ and conditional probabilities $P(x_i{}^{(j)} \mid y_i)$ are calculated through a frequency count or a maximum likelihood estimation (MLE)[20]. For

[16] *https:// en.wikipedia.org/ wiki/ Kernel_ method*

[17] *https:// en.wikipedia.org/ wiki/ Radial_ basis_ function_ kernel*

[18] Refer to *https:// scikit-learn.org/ stable/ modules/ naive_ bayes.html* for more information and examples.

[19] *https:// en.wikipedia.org/ wiki/ Bayes|%27_ theorem*

[20] More information on MLE can be found at *https:// en.wikipedia.org/ wiki/ Maximum_ likelihood_ estimation*

a given input vector x, predicted class is the one for which the Posteriori probability is maximum [Bonaccorso, 2018].

Multinomial Naive Bayes uses multinomial distribution for modelling which is useful when each value of a feature vector represents the number of occurrences of a term or relative frequency [Bonaccorso, 2018], like in a text-based classification [Rennie et al., 2003].

If there is a feature vector $x = (x_1, ..., x_n)$ having n features and x_i represents frequency, then the likelihood of observing x is given by[21]:

$$p(\mathbf{x} \mid C_k) = \frac{(\sum_i x_i)!}{\prod_i x_i!} \prod_i {p_{ki}}^{x_i}$$

If a particular class and feature value never occur together in the training data, then probability estimate based on the frequency will be zero. Multinomial Naive Bayes thus requires a correction parameter called pseudocount, such that no probability is ever set to be zero. Since multiplying with a 0 wipes out all other information in rest of the features. This regularization is called Laplace smoothing when the pseudocount is one and Lidstone smoothing in general case. The advantage of Naive Bayes classifiers is that they work well even with a fewer set of training instances.

K-Nearest Neighbor (KNN)

K-Nearest Neighbor Classifier is a non-parametric, instance-based learning algorithm which computes the classification based on the majority vote of the k nearest neighbor of each query instance where k is specified by the user. Output of the KNN classifier is a class membership. The value of k is dependent on the data; large values of k reduces noise but makes boundaries less disntint. Figure A.15 illustrates a simple classfication task of a KNN classifier[22].

Following are the steps involved in KNN classification:

1. For a given query instance in the feature space, KNN identifies the k nearest neighbors based on a distance metric (e.g., Euclidean distance) In the above example with k=3, I identify three instances as nearest neighbors: two red triangles and one blue square.

2. KNN takes a majority vote of the classes of the k nearest neighbors. In the example, red triangle has majority vote of two.

3. KNN assigns the majority vote as the class of the query instance. In the example, query instance is assigned the class: red triangle.

KNN classifiers perform both binary and multi class classification and are very good for outlier detection.

[21] Formula taken from *https://en.wikipedia.org/wiki/Naive_Bayes_classifier#Multinomial_naive_Bayes*

[22] *https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm#/media/File:KnnClassification.svg*

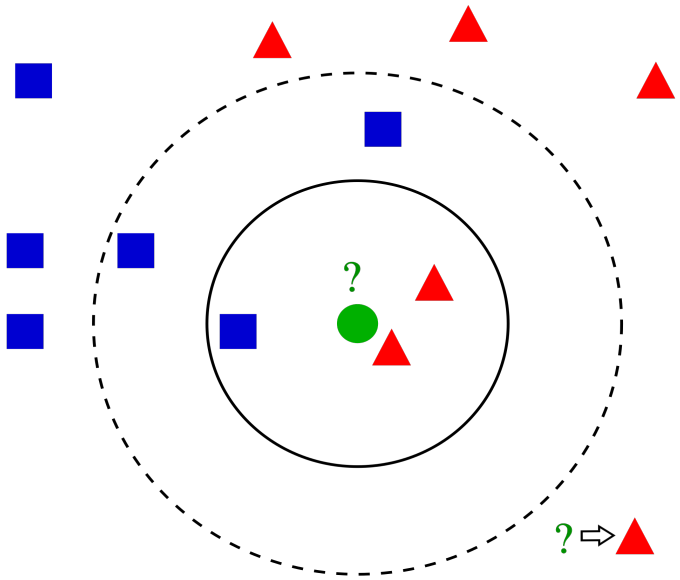

Figure A.15: A simple example of a KNN Classifier

Logistic Regresssion

Logistic Regression is a probabilisti model used for classification which works on linearly separable classes. It can extend to multiclass problem by using the OnevsRest strategy (see Section A.5.4). Logistic regression predicts the probability of an instance belonging to a certain class. It is calculated from the logistic function which is a sigmoid function taking any real value as an input and produces 0 or 1 as an ouput. It is an inverse form of logit function or log of odds ratio (odds in favour of a particular event) [Raschka, 2015].

Logit function is denoted as $\text{logit}(p(x)) = \ln\left(\frac{p(x)}{1-p(x)}\right) = \beta_0 + \beta_1 x$ where $p(x)$ is the probability of a particular class happening. Taking exponentiation on both sides:

$$\frac{p(x)}{1-p(x)} = e^{\beta_0+\beta_1 x}.$$

In classification, $\text{logit}(p(y = 1/x)) = e^{\beta_0+\beta_1 x}$ is the conditional probability that a particular instance belongs to class 1 where x are features.

Taking inverse, logistic function is written as $p(x) = \frac{1}{1+e^{-(\beta_0+\beta_1 x)}}$, where $p(x)$ is the probability of the instance belonging to one of the two classes in case of binary classification. $\beta's$ are the parameters to be learned.

A simple logistic regression classifier is illustrated in Figure A.16. It shows the prediction of target class based on petal length and width.

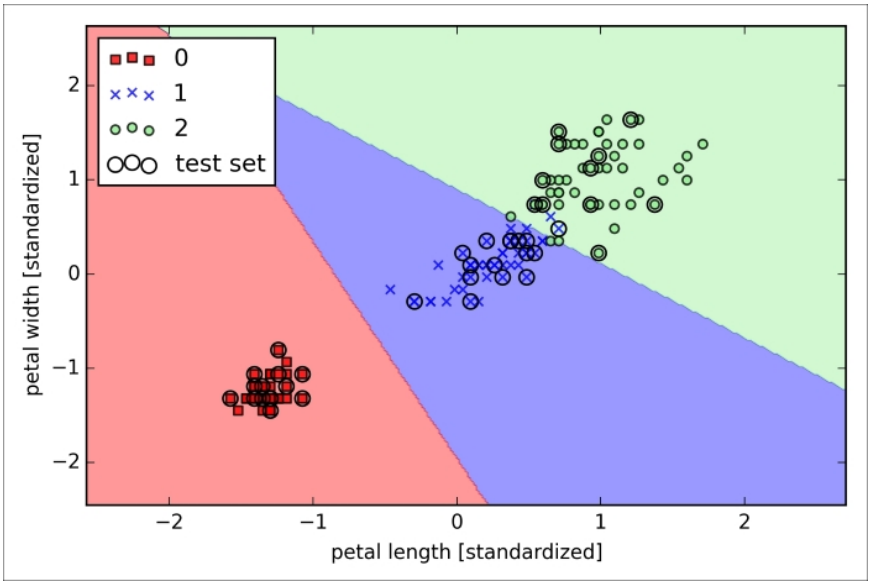

Figure A.16: An example of logistic regression classification

Multilayer Perceptron (MLP)

MLP is a multi-layer feed forward artificial neural network consists of atleast three layers: input layer, hidden layer and an ouput layer. If an MLP network contains multiple hidden layers, then it is also called deep neural network. Each node except the nodes of input layer is called a neuron in an artificial neural network. Figure A.17 shows the building blocks of a neural network [Bonaccorso, 2018]. Each neuron uses a non-linear activation function whose parameters are optimized through a technique called Backpropogation (see [Rumelhart et al., 1986a] and [Rumelhart et al., 1986b] for more details). Logistic (sigmoid), hyperbolic tangent (tanh) and rectified linear units (ReLU) are the commonly used non-linear activation functions.

$$\textbf{sigmoid } g(z) = 1/(1+e^{-z})$$

$$\textbf{tanh } g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

$$\textbf{ReLU } g(z) = z^{+} = \max(0, z)$$

An illustration of a fully connected MLP with one hidden layer (shallow/vanilla neural network) is shown in Figure A.18. A fully connected neural network means every neuron in a layer is connected to all the neurons in the next layer. $_{i}th$ activation function in an ^{l}th layer is denote by $a_i^{(l)}$. $w_{jk}{}^{(l)}$ denotes the weight from $_{j}th$ input to $_{k}th$ activation function of ^{l} th layer.

The steps in a MLP is as follows:

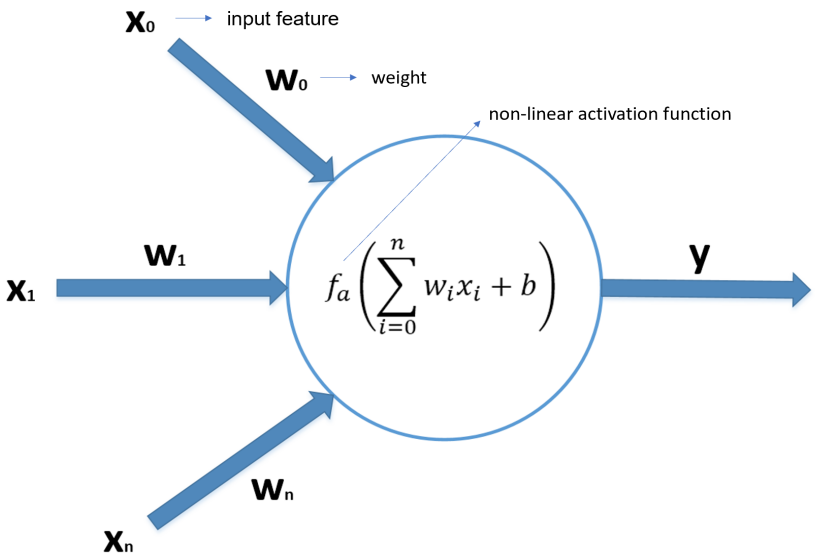

Figure A.17: A neuron in an artifical neural network

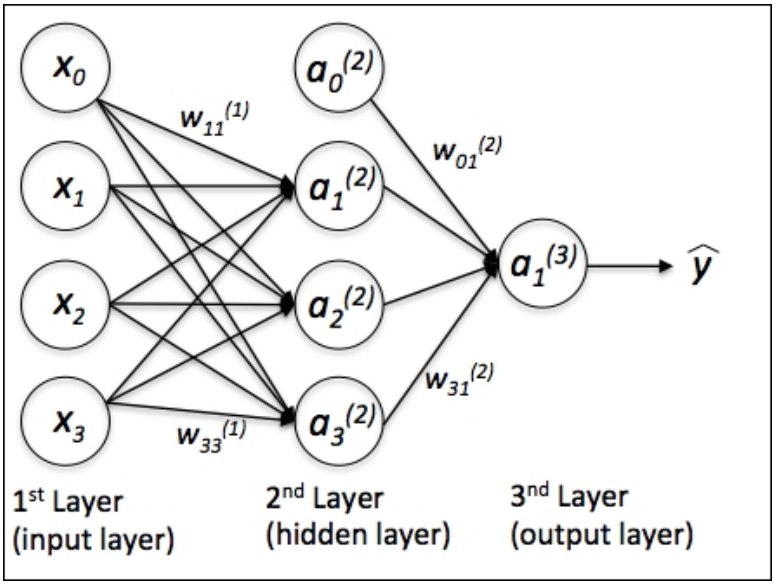

Figure A.18: Illustration of a MLP artificial neural network

1. The input layer denotes the inputs (x_0, x_1, x_2, x_3)[23]. In a text-based classification, this can be one-hot coded or tf-idf [24] transformed vectors. Weights are randomly initialized[25] and activation functions are decided before-hand. The input fed through the network produces a predicted output $\hat{y}$.

2. From the network's ouput, error is calculated $(y - \hat{y})$ using an objective cost function (e.g. logistic function).

3. The error is backpropagated through the network to compute gradients with respect to the weights in each layers.

4. The weights are updated based on the computer gradient using stochastic gradient descent method[26].

5. Steps 1-3 are repeated for multiple epochs (set as hyperparameter) to learn the weights.

6. At the end of the final epoch, an output 1 or 0 is predicted for binary class. (In multiclass classification, a softmax function $\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{l=1}^{k} \exp(z_l)}$ is used an acitivation in the output layer instead of lgoistic or tanh or ReLU acitivation function thus resulting in 1 for predicted class and others 0).

MLP is a supervised learning technique and performs multiclass classification using OnevsRest strategy (see Section A.5.4). It works well on any kind of data, thus making it attractive for especially non-linear data.

Decision Trees

Decision tree is yet another popular and one of the interpretable classifiers. They are tree structures with leaves representing class labels, nodes representing features and branches represent conjunctions of features that lead to those class labels[27]. Decision tree classifiers predicts the target class by asking a series of questions. It recursively splits the data into subsubsets based on a feature value. At every split, the feature that results in the highest information gain (IG)[28] is chosen. The recursive ends when a subsubset of instances at a given node all belongs to the same target class (all the leaves of the decision tree are pure) [Raschka, 2015]. In simpler terms, at every point, a feature is chosen and its value determines the subsubsets. This process is repeated until all the instances in a subsubset belong to the same class. Generally, a very deep tree with many nodes is generated as a result of multitude of features that are present in many of the machine learning problems. This leads to overfitting and thus requires pruning of the tree by

[23] x_0 is a bias term to avoid zero input to the network
[24] *https://en.wikipedia.org/wiki/Tf-idf*
[25] Different methods to initialize weights exist.
[26] Refer to *https://scikit-learn.org/stable/modules/sgd.html* for more information and examples on Stochastic Gradient Descent
[27] *https://en.wikipedia.org/wiki/Decision_tree_learning*
[28] *https://en.wikipedia.org/wiki/Information_gain_in_decision_trees*

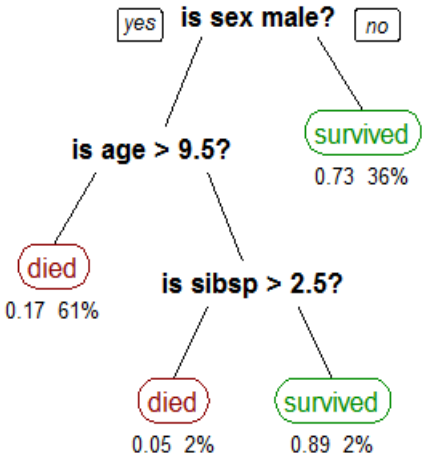

Figure A.19: A simple example of a Decision Tree Classifier

setting a limit for maximal depth of the tree [Raschka, 2015]. Figure A.19 illustrates a simple decision tree classifier showing survival of passangers in Titanic ('sibsp' denotes of number of siblings/spouses on board)[29].

Random Forest

Random Forest classifiers is an ensemble learning method, learning from an ensemble of decision trees. An ensemble learning methods combines weak learners (slightly correlated with true classification than a random guess) to build a strong learner (strongly correlated with true classification) to provide robust method. In essence, Random Forest classifiers builds a multitude of decision trees (number of trees 'k' can be set as a hyperparameter) and predicts the target label as the mode of the classes predicted by the ensemble[30]. Each decision tree in the ensemble learns from a random 'd' set of features (whereas in Decision Tree method, the model evaluates all the features) [Raschka, 2015]. The algorithm follows the below steps:

1. Draw a random sample of size n from training set with replacement.
2. Build a decision tree from the sample. At each node:
   a) Randomly select d features without replacement.
   b) Split the node into subsubsets using the feature providing the best split (An example objective function: maximizing the information gain).

[29] *https://en.wikipedia.org/wiki/Decision_tree_learning*
[30] *https://en.wikipedia.org/wiki/Random_forest*

3. Repeat the steps 1 to 2 k times.

4. Assign the class label by a majority vote (mode) of the predictions by decision trees.

While Random Forest do not provide the same interpretability of the model as a decision tree due to their complex 'forest' like structure, they are less prone to overfitting.

Gradient Boosting

Gradient Boosting is another popular ensemble learning method, like Random Forest classifiers, also learns form an ensemble of decision trees. Gradient Boosting [Friedman, 2001] builds a tree ensemble step by step (forward stage-wise additive modelling) [E. Schapire, 2013] [Bonaccorso, 2018]. At each step, the algorithm fits a decision tree using a weighted version of the data: increasing the weight of the misclassified instances and decreasing the weight of the correctly classified instances in the previous step. Thus, the future weak focus on misclassified instances (Boosting [E. Schapire, 2002]). The goal is to minimize the target loss function using Steepest Gradient Descent[31].

Gradient Boosting methods requries less training data and fewer features to achieve the same performance when compared to training separately.

A.5.3 Multiclass and Multilabel Classification

For classification of code cells, I employed two strategies: Multiclass and Multilabel Classification. Classification tasks can be either binary-class or multi-class problem. Binary classification is used when an instance is classified into one of the two classes. Multiclass or Multinomial classification is used when an instance is classifed into one of the three or more classes. Multilabel classification is the problem of predicting multiple labels for each instance. As data science steps classification contains more than two classes and each code cell can have either a single label or multiple labels, I investigated both multiclass classification and mutilabel classification. In this section, I briefly summarise the techniques, classification algorithms and the reasoning for choosing them. Experiments on the dataset are discussed in Section 5. Results and evaluation are discussed in Section **??** and Section 5.7

Multiclass Classification

Multilabel classification assigns a set of labels to each instance. For example, in the dataset, if a code cell perform the following actions: loading multiple data files and describe the data, it should be classified as: load_data. For this, I will train the classifier

[31] *https://en.wikipedia.org/wiki/Gradient_descent*

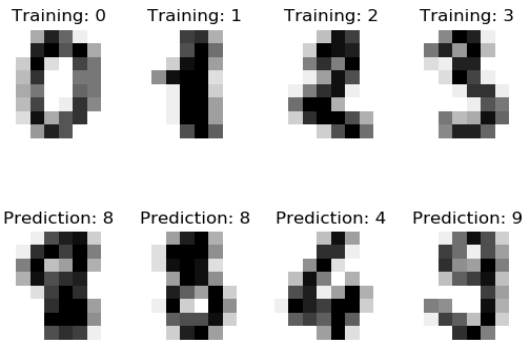

Figure A.20: Multiclass Classification of Digits

with the primary label assigned to each instance as discussed in the Section 5.5.1. Figure A.20 illustrates the multiclass classification of digits[32].

Multilabel Classification

Multilabel classification assigns a set of labels to each instance. For example, in the dataset, if a code cell perform the following actions: loading multiple data files and describe the data, the classifier assigns labels: [load_data, data_exploration]. For this, I will train the classifier with all the relevant labels assigned to each instance as discussed in the Section 5.5.2. Figure A.21[33] illustrates a multi label classification where some of the instances are assigned two class labels.

A.5.4 Classification Strategies

Binary classifiers can be extended to multiclass problems by using one of the two important classification strategies: OnevsOne and OnevsRest. OvO and OvR transforms the multiclass problem into multiple binary class problems.

OnevsOne (OvO)

In OvO, one binary classifier per pair of classes is trained. Each classifier uses the instances from the pair of classes in the training set to learn. For an unseen instance,

[32] Sklearn Multiclass Classification Example: Recognizing hand-written digits *https://scikit-learn.org/stable/auto_examples/classification/plot_digits_classification.html#sphx-glr-auto-examples-classification-plot-digits-classification-py*

[33] *https://scikit-learn.org/stable/auto_examples/plot_multilabel.html#sphx-glr-auto-examples-plot-multilabel-py*

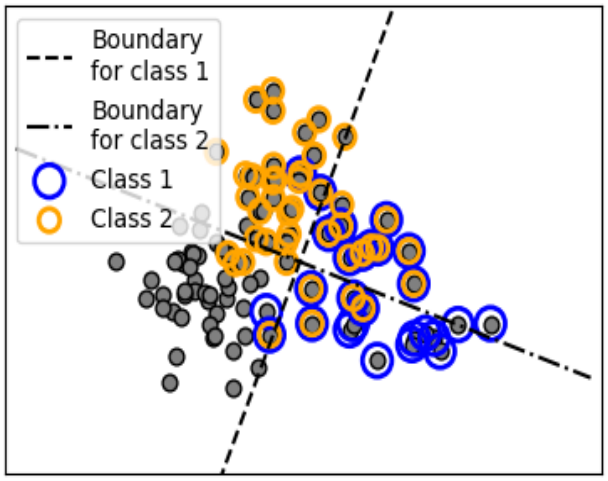

Figure A.21: Multilabel Classification

a majority voting is applied to predict the class from the combined classifiers[34]. In total $n * (n-1)/2$ classifiers are trained for a multiclass problem with n classes. The complexity is quadratic $\mathcal{O}(n^2)$ in OvO strategy and so, is not first preferred strategy [Bonaccorso, 2018].

OnevsRest (OvR)

In OvR, one binary classifier per class is trained, where the particular class is treated as the positive class and the rest are treated as the negative class. Thus, I will train n binary classifiers if I have n labels in the multiclass problem. It has linear complexity $\mathcal{O}(n)$ and is the default choice of strategy. OvR can suffer from imbalanced distribution of the dataset since prediction can be skewed towards the class with more samples but provides more interpretability[35]. OvR also supports Multilabel classification.

A.5.5 Evaluation Metrics

In this section, I will give brief information about the evaluation metrics [Powers, 2011] that are considered in this thesis.

Accuracy

Accuracy is the fraction of predictions the model got right. Accuracy is defined as:

$$\text{Accuracy} = \text{Number of correct predictions}/\text{Total number of predictions}$$

[34] Refer to *https://scikit-learn.org/stable/modules/multiclass.html#one-vs-one* for more information on OvO

[35] Refer to *https://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html* for more information and examples

In binary and multiclass classification, accuracy in scikit-learn is equal to the Jaccard index[36], or Jaccard similarity coefficient, defined as the "size of the intersection divided by the size of the union of two label sets, is used to compare set of predicted labels for a sample to the corresponding set of labels in y_true". In multilabel classification, accuracy in scikit-learn computes subsubset accuracy: the set of labels predicted for a given instance "must exactly match the corresponding set of labels in y_true"[37].

Precision

Precision is defined as the number of true positives TP over the number of true positives plus the number of false positives FP.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Recall

Recall is defined as the number of true positives TP over the number of true positives plus the number of false negatives FN.

$$\text{NPV} = \frac{\text{TN}}{\text{TN} + \text{FN}}$$

Confusion Matrix

A confusion matrix allows visualization of the performance of an algorithm with regards to false positives, false negatives, true positives, and true negatives. Each row of the matrix represents the instances in a predicted class while each column represents the instances in an actual class[38]. Figure A.22 illustrates a simple confusion matrix [Raschka, 2015].

F1-score

F1 score is the harmonic mean of precision and recall.

$$F_1 = 2 \cdot \frac{\text{PPV} \cdot \text{TPR}}{\text{PPV} + \text{TPR}} = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}}$$

[36] *https://en.wikipedia.org/wiki/Jaccard_index*
[37] Refer to *https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html* for more information and examples
[38] *https://en.wikipedia.org/wiki/Confusion_matrix*

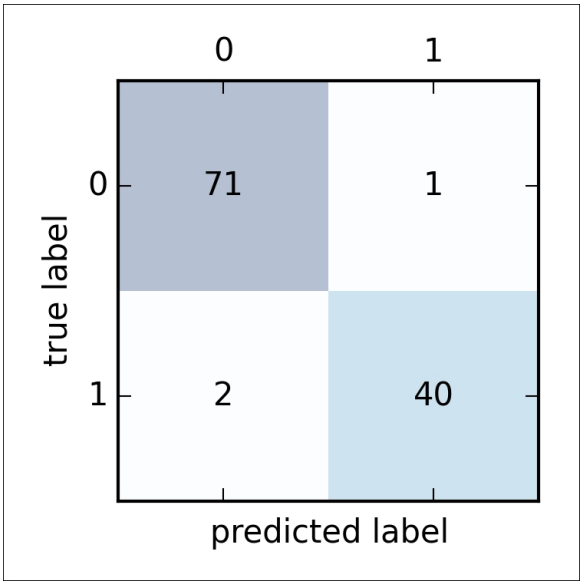

Figure A.22: A simple confusion matrix