Jupyter Notebook Search Engine

Michael L. de Jong

Layout: typeset by the author using LaTeX.
Cover illustration: Google datacenter Douglas County by Google

Jupyter Notebook Search Engine

Michael L. de Jong
11246618

Bachelor thesis
Credits: 18 EC

Bachelor *Kunstmatige Intelligentie*

University of Amsterdam
Faculty of Science
Science Park 904
1098 XH Amsterdam

Supervisor
Dr. M.J. Marx

Information and language processing systems (ILPS)
Informatics Institute
University of Amsterdam
Science Park 904
1098 XH Amsterdam

June 26st, 2020

Contents

Abstract

This thesis describes a way to implement a search engine of Jupyter notebooks using a dataset from Adam Rule. This includes the way the data is preprocessed, indexed into Elasticsearch indices and the removal of empty, duplicate and backup notebooks. The effect on the results of the data of Rule was negligible. of those removal was It also shows how the Search Engine Result Page was produced and how the indexed documents were ranked. This is done by a code/text classifier that was trained using samples of different notebook cells. Finally the classifier is tested and the relevance of the returned documents is calculated.
keywords: Search Engine, SERP, code/text classification, TF-IDF, Logistic Regression, duplicate detection

Chapter 1

Introduction

There are over 6 million Jupyter notebooks on Github and that total is ever growing. There is however, not a good way to search trough them to be able to find their desired part of the notebooks. The main goal of the project is to make a search engine that is aware of the Jupyter notebook format so that it improve the search. The Jupyter notebook format is a structured JSON file. In these files, there are multiple types of information in different cells. Thus the search engine has to search the different cell types that can exist inside notebooks. If combined, it might be laborious to, for example, separate a notebook about Shakespeare text learning and a notebook that solely contains the Shakespeare poëms. Hence is searching for a relevant Jupyter notebook more viable when the search engine is aware of the different cell types? To answer this question the following sub questions arise.

1. How do different indices influence evaluation scores?
2. How can we discover duplicate notebooks and how frequent are they?
3. Does removing notebook files change the outcomes of the research by Rule?
4. What queries and information needs will be used to to evaluate the different indices?

To achieve this, there exists dataset from Adam Rule 2018 and a well known search engine available to download which will be used, namely Elasticsearch Documents can be added to an index, which is a collection of documents that are related to each other. Elasticsearch stores data as JSON documents. Each document correlates a set of keys (names of fields or properties) with their corresponding values (strings, numbers, Booleans, or other data types). Elasticsearch uses an inverted index, which is designed to allow full-text searches. An inverted index lists every unique word that appears in any document and identifies all of the documents each word occurs in. Interaction with Elasticsearch can be done in Python and Kibana, which is a management program. These tools will be used for this project.

Chapter 2

Theoretical foundation

On the subject of Jupyter notebooks, several people investigated the notebooks available on Github, namely in Exploration and Explanation in Computational Notebooks by Adam Rule 2018 and the follow up by Jenna Landy 2019.

Adam Rule Rule 2018 collected around 1,25 million notebook documents and performed three studies on the collected notebooks. In the first, they analyzed the collected notebooks on content. In a second study, they examined over 200 academic notebooks, finding that most described methods, while only a minority included reasoning or results. In a third study, They interviewed academic data analysts that considered computational notebooks personal, exploratory, and messy and used other media to share their analyses. The research concluded that These studies demonstrated that there is tension between exploration and explanation in the construction and sharing of computational notebooks.

The first study on notebook content was expanded upon by Jenna Landy 2019 with about 4 million notebooks. This was 93% of the non forked notebooks at the time. This exploration included the popularity of imports, exactly what kind of visualisations were used and what packages are included in the notebooks and other factors such as errors in the notebooks.

In the field of search engines, a Jupyter Notebook search engine was also developed. In Grappiolo et al. 2019 they implemented the "Semantic Snake Charmer" or SSC, which is a domain knowledge-based search engine for Jupyter Notebooks. They composed the SSC out of three modules: (1) a human-machine cooperative module to identify internal documentation which contains the most relevant domain knowledge, (2) a natural language processing module capable of transforming relevant documentation into several semantic graph types, (3) a reinforcement learning based search engine which learns, given user feedback, the best mapping between input queries and semantic graph type to rely on. This implementation is in use at Océ Technologies as internal search engine for Jupyter notebooks and thus not open source.

To build the search engine the method described by Buck and Koehn 2016 can be used. In this paper, they show, among other elements, that using the cosine distance of tf/idf weighted document vectors already provides a quick and reliable way to align documents.

This approach is also used in Elasticsearch as the vector space model, which together with a relevance scoring function are used to calculate relevance scores. These scores are then used to rank the documents returned by elasticsearch. The scoring function used to calculate relevance in Elasticsearch is called the Lucene practical scoring function. Lucene is the open-source search engine that Elasticsearch is based upon. The way this is done is shown in figure 2.1 below.

```
score(query,document) =
    Normalise query with query
    Apply the coordination factor to query and document
    Calculate the term frequency in document
    Calculate the inverse document frequency for term
    Apply query tuning
    Normalise document with the inverse square root of the number of terms
```

Figure 2.1: Lucene scoring function

Chapter 3

Data quality

The data for this project originates from Adam Rule Rule 2018 which were 6 zipfiles of around 50gb which included notebooks labeled with a number followed by the file extention of notebooks: The first documents is called "*nb*_0.*ipynb*" as an example. A comma separated file (csv) came With the zipfiles and contains metadata including the original name of the notebook, the html url, the filesizes and information about the repository it came from.

The Jupyter Notebook documents contains the inputs and outputs of a interactive session as well as additional text that accompanies the code but is not meant for execution. In this way, notebook files can serve as a complete computational record of a session, interleaving executable code with explanatory text, mathematics, and rich representations of resulting objects. These documents are internally JSON files and are saved with the .ipynb extension. Since JSON is a plain text format, they can be version-controlled and shared with colleagues.

It contains two separate cell types, namely markdown and code cells.

Figure 3.1 below contains the cell statistics researched by Rule. The right table in the figure shows that notebooks often start with markdown and use code cells later in the file.

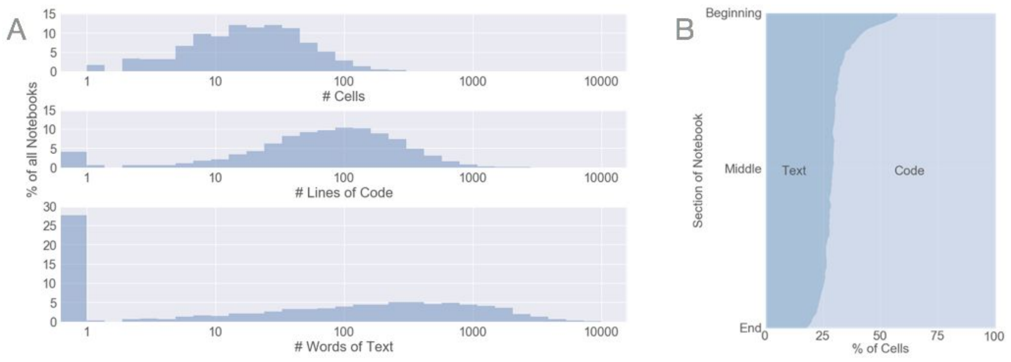


Figure 3.1: Rule cell statistics
In table A, the length of the notebook cell, code and markdown is measured. In table B the average location of markdown and code of the notebook files is measured.

Rule also mentions additional statistics: 2.2% of the notebooks has no code en 27.6% has no text. 40 different programming languages are used. In 85.1% of the notebooks the programming language is mentioned, almost all are in Python, 96.3% (Python 2.7 is 52.5%). 1% of the notebooks is written in R en Julia. In 89.1% of the R en Julia notebooks imports were used.

The most used Python imports are Numpy (67.3% of the Python notebooks), matplotlib (52.1%), and pandas (42.3%). These packages are largely used to create visualisations and are used in data management. From this you could conclude that notebooks are used for visualisations, for example in data science.

3.1 Data oddities

The following characteristics were found in the dataset from Adam Rule. The first was the inclusion of outdated notebook files, the second the "checkpoint" files, the third was the empty files and the last were the duplicate notebook files. Documents that belong to one of these four categories were removed from the indexed documents. The outdated notebook files were files that used a different format to the current Jupyter notebook file structure. To make preprocessing of every file quicker these notebook files were removed. Of the indexed files 232785 of them are Jupyter notebook checkpoint files, which are backup files of other files. From the 1,25mil to 1,213,723 indexed documents and without checkpoints 980,937. Landy 2019 also removed these backup files to avoid duplicates.

These checkpoint files are created when you run and save a notebook. A lot of the most popular names can be traced back to kaggle datasets or to simple algorithms like KNN and SVM. These files were removed with the query from figure

3.2. The wildcard query matches with every name that includes "-checkpoint" which is always included in the name if it is a checkpoint file.

```
POST /info_clone/_delete_by_query
{
  "query": {
    "wildcard": {
      "name": {
        "value": "*-checkpoint*"
      }
    }
  }
}
```

Figure 3.2: removal of checkpoint files

The third point is empty files, files completely without code or markdown cells. After the removal of the checkpoint files and the outdated notebook files, there were 1840 empty notebooks left. These empty notebooks were deleted as described in the Methods. After these files were eliminated there were 979,097 indexed documents left in the index.

Lastly the duplicate files. When testing the search engine there were an abundance of files that were seemingly the same. The duplicates were removed by mapping every document to a hash value and comparing them. When files map to the same hash value there might be a duplicate file. The way this was done is further explained in Methods. In Rule his dataset there were 234,472 duplicate files.

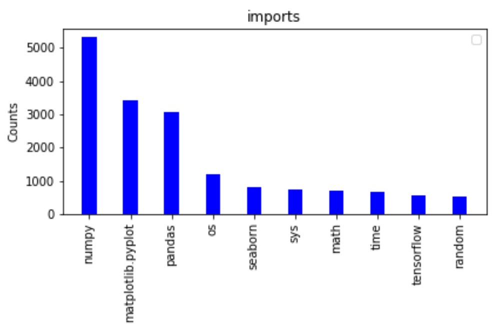
After the removal of these files 744,625 indexed documents were left of the original total of 1.25 million notebooks from Rule. The amounts mentioned in this section are summarised in table 3.3 on the next page.

Data part	Number	Percentage
Rule total	1,227,573	100
Indexed total	1,213,723	98.89
Indexed remaining total	744,625	60.66
Total removed from index	469,097	38.21
Total removed from Rule	482,947	39.34
Outdated files	13,850	1.13
Checkpoint files	232,785	18.96
Empty files	1840	0.15
Duplicate files	234,472	19.10

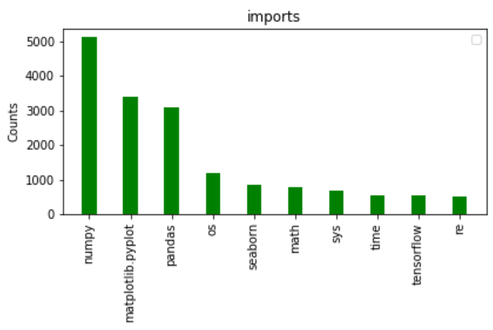
Figure 3.3: Data parts in relation to the Rule dataset

3.2 Comparison

As a result of the removal of over 450 thousand files the original numbers from Rule might have been altered. So in this section, the visualisations were created to examine any changes to the composition of the data. Namely the imports, names, languages, classes and functions.



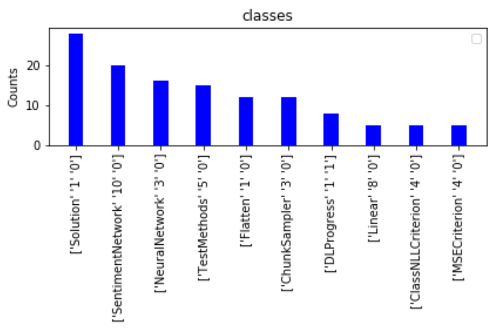
(a) Before removal



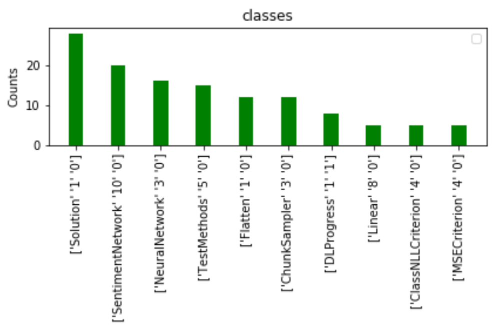
(b) After removal

Figure 3.4: Top 10 Python imports of all indexed documents

The ratio of the top 10 classes and functions, as depicted in figure 3.5 and 3.6 respectably, stayed the same. Variations of f is used a lot for functions and forms of solutions or network names for classes.

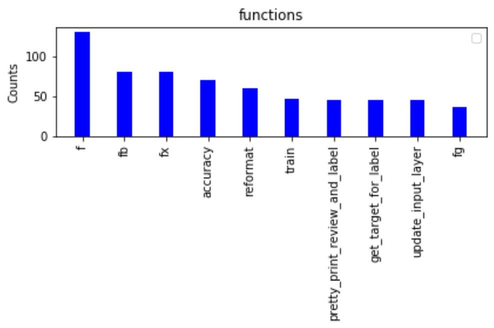


(a) Before removal

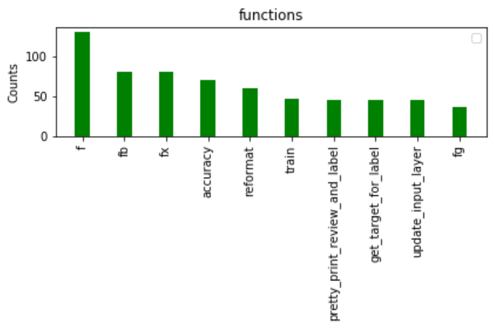


(b) After removal

Figure 3.5: Top 10 Python classes before and after removal



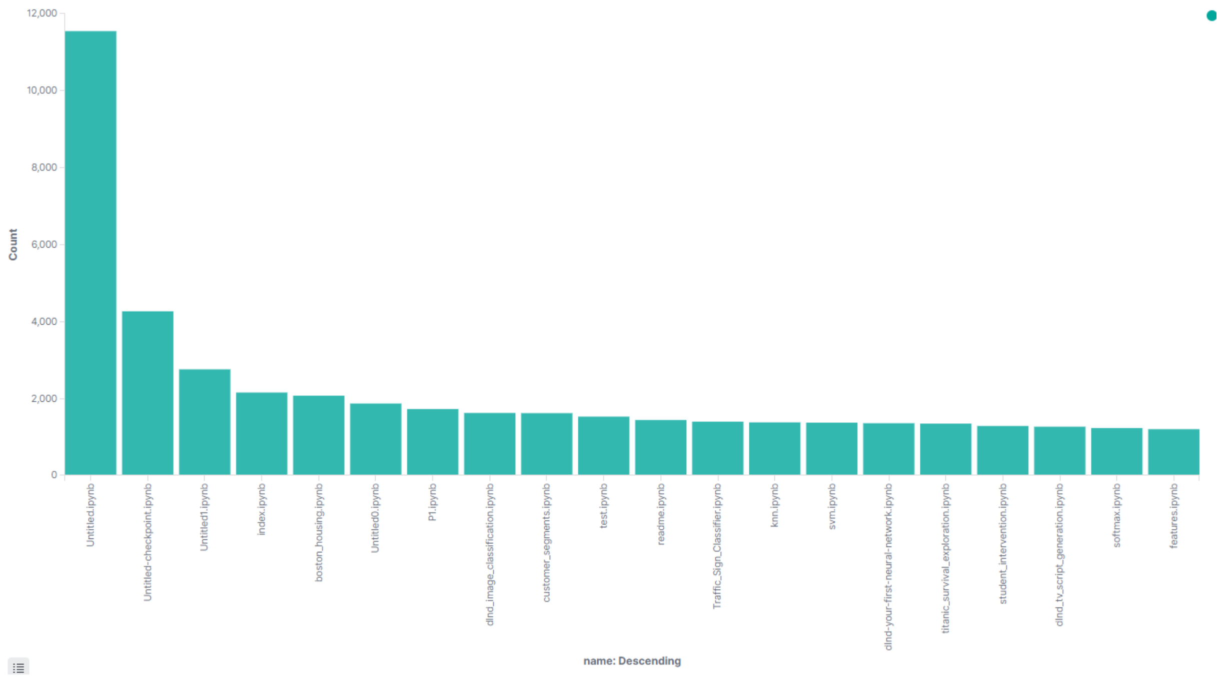
(a) Before removal



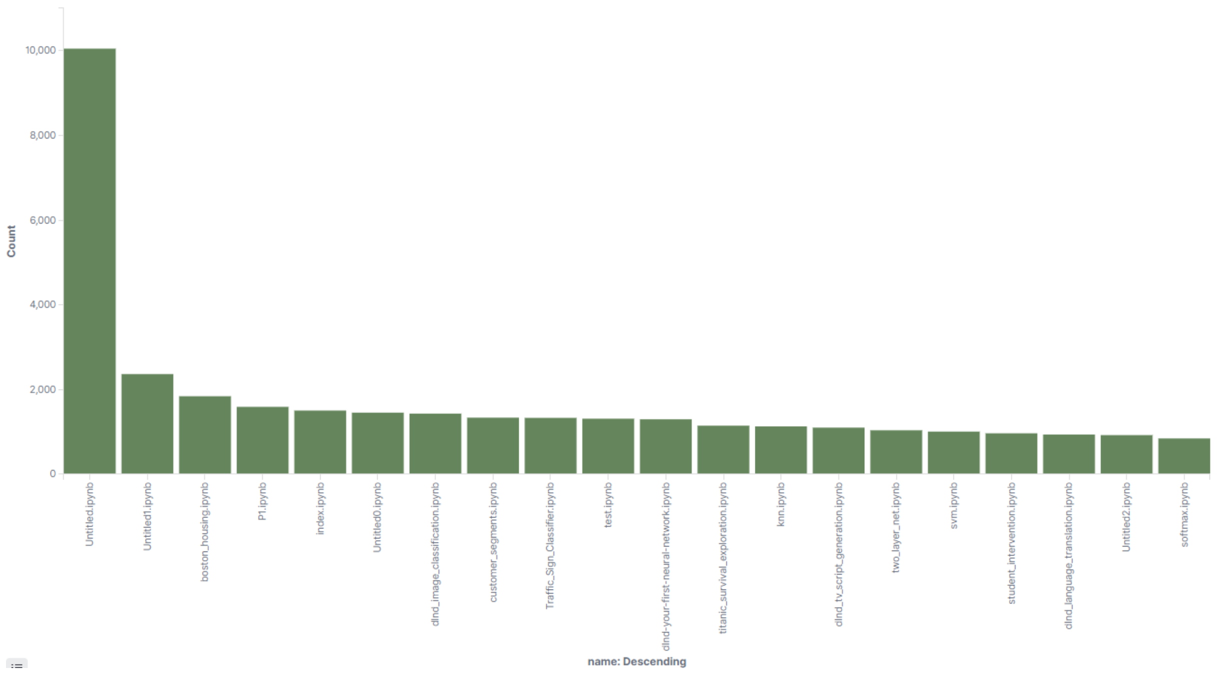
(b) After removal

Figure 3.6: Top 10 Python functions before and after removal

In comparison of the rule data of the top 10 imports in figure 3.4 above, the top 5 Python imports were equal. The top 10 however, was affected by the removal. The most frequent names of the notebooks in figure 3.7 below did not include the checkpoint files anymore because those were removed and the name Untitled2 was no longer in the top 20. Notebooks with specific names however, usually contained different code or markdown, so these are still in the top 20. Lastly, the different programming languages used as shown in figure 3.8, the unknown category has decreased by a percent while the percentage of other categories were comparable from before the removal.

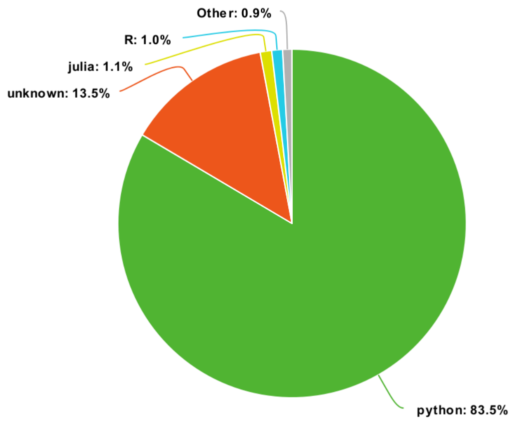


(a) Before removal

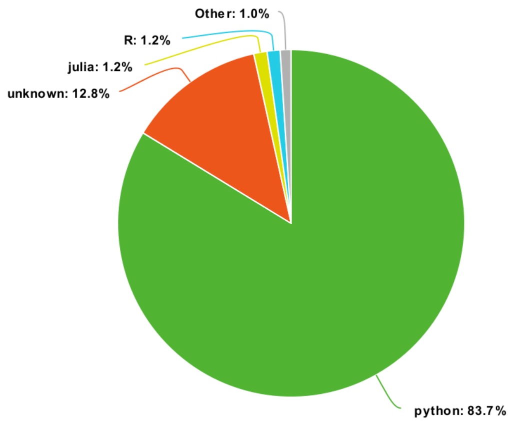


(b) After removal

Figure 3.7: Top 20 names of all indexed documents



(a) Before removal



(b) After removal

Figure 3.8: Languages before and after removal

Overall, the visualisations indicate that the removal of the over 450 thousand notebooks had little effect on the ordering of the most frequents imports, names,116languages, classes and functions used by the creators of the notebooks. This answers sub-question 3: Removing notebook files does not change the outcomes of the research by Rule.

Chapter 4

Methods

4.1 Data handling

To create the search engine the following items had to be completed: Preprocessing, the indexing, the search result page, the removal of certain files and the Ranking method. This section describes the implementation of those items.

A single computer was used to run all code for this project. To put the execution times mentioned into perspective, the hardware includes a AMD Ryzen 1500X quad-core processor with 8 threads coupled with 16GB 2666 Mhz RAM and a Nvidia GTX 1070 graphics card. The Elasticsearch indices were created and stored on a 7200RPM hard drive. The hard drive was the limiting factor because of the get requests of Elasticsearch.

4.1.1 Preprocessing

The rule data contains all the raw notebooks and files with metadata so the first necessary step is to do preprocessing. This is done by using a Python script to separate the notebooks into different information categories. A total of 7 different indices which were created, namely info, markdown, code, comments (simple) and comments2, together with two backups of info index. The csv datafile which contains metadata of all notebooks, including the notebook id, URL and name. Other fields like filesize were dropped from this index. The first processing step is to create a list of all ids and zipobjects is generated so that the zipfiles don't have to be unzipped. This was done because unzipping would require more then 300GB of available storage on the hard drive. The oldest Jupyter notebooks (v1 and v2) have a different structure to more recent versions of the notebook format, so the choice was made to refrain from indexing them as explained in section 3. After that the notebook language is extracted and the code and markdown are separated. The comments of the code are also extracted in a separate column. This is done in a simple way that just finds lines that include "#" which is used as a comment indicator in Python and R. The different parts are then added to a pandas dataframe. The pseudocode of the zip extraction is shown in figure 4.1 below. When this is done for every file in the zip the dataframe is exported to a pickle file, which is a Python object serialization file. This is done so the zipfiles do not have to be used anymore. The process to generate the pickle files took on average 5.5 hours to run, with the largest Zip taking just over seven hours to process.

After generating the pickle files, the source code cells needs to be further processed. For this, code of Jenna Landy 2019 has been used to separate imports, functions, classes and comments from Python, R and Julia. These comments take into account all comment types of the three languages.

```
def get_text_zip():
    with open the zip:
        read the notebook
    if there are notebook cells:
            get the kernel programming language
            get the markdown cells
            get the code cells
        for cell in markdown cells:
            if source exists in cell:
                append to markdown
        for cell in code cells
            if source exists in cell:
                append to code
                get comments from code
        return the language, markdown, comments and code
```

Figure 4.1: Retrieve data from notebooks pseudocode

The 1,25 million documents were compacted into 7 Elasticsearch indices, as shown in table 4.1 below. The different methods of comment extraction gave the same size index. The information (info) index has been cloned before major removals of files as a backup.

Index	Document count	Storage size
code	1213723	5.7 GB
comments	1213723	2.4 GB
comments2	1213723	2.4 GB
info	1213723	452.6 MB
info_clone	980937	446.9 MB
info_clone2	744625	358.7 MB
markdown	1213723	3.4 GB

Table 4.1: Elasticsearch indices and sizes

4.1.2 To Elasticsearch

When all the notebook files are processed they can be uploaded to Elasticsearch. The different information is put in several different indices, namely the information, code, markdown, comments and comments2 index. Comments2 because that used the more intricate comment extraction method from Jenna.

Elasticsearch needs indices to fill which follow a certain index pattern or mapping. figure 4.2 below shows the mapping used for the information index. It needs the id of the notebook, the URL, name and programming language of the files which were available from the preprocessing step. Elasticsearch lookup can be an expensive operation depending on query size, so shards were used. The number of shards help improve the search speed inside of Elasticsearch by separating the documents in chunks that are hence called shards.

```
# Initialise the elastic search index
init_info_index = {
    "settings" : {
        "number_of_shards": 5,
        "number_of_replicas": 0,
    },
    'mappings': {
            'properties': {
                'nb_id': {'type': 'integer'},
                'html_url': {'type': 'keyword'},
                'name': {'type': 'keyword'},
                'language': {'type': 'keyword'},
        }
    }
}
es.indices.create(index = 'info', body = init_info_index)
```

Figure 4.2: Initialisation of the info index

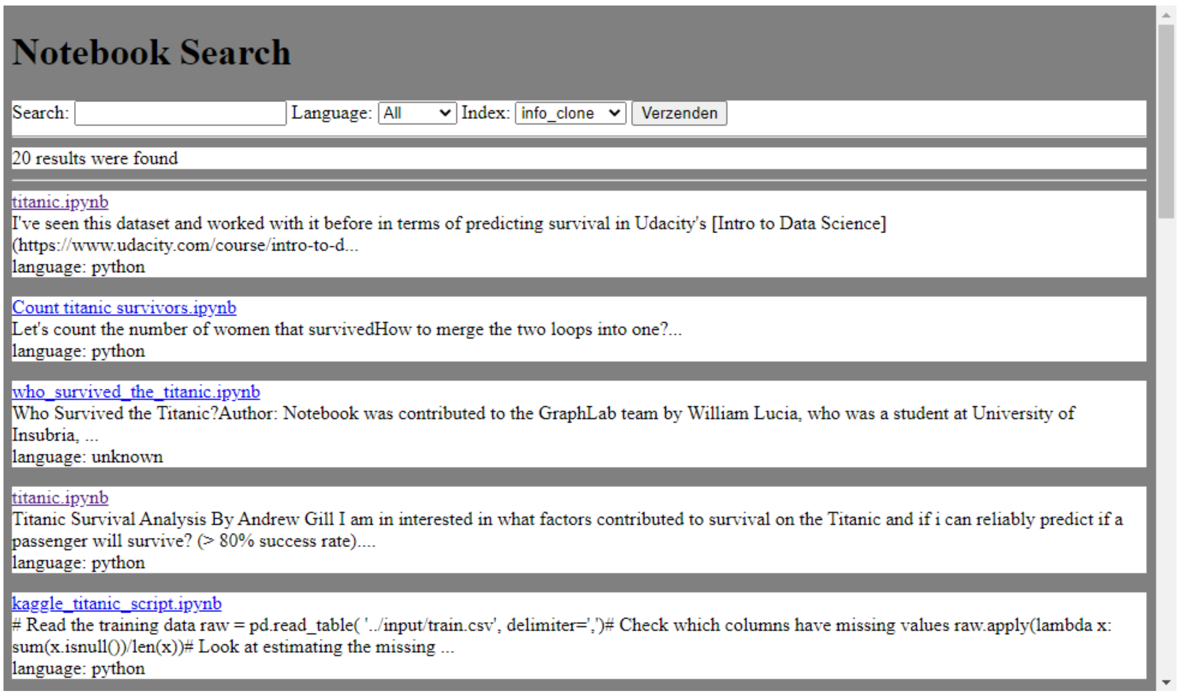
To upload all rows of the pandas dataframe to Elasticsearch a way other than iterative was required, it took too long to update the index using an iterative method, The way this was accomplished was to use the pseudocode method described in figure 4.3 underneath. Elasticsearch includes parallel functions which can be used to index documents in bulk using a generator. The reason this approach is faster is rather than processing a single notebook document at a time, with the generator the processor can process multiple documents at the same time and Elasticsearch can input those documents straight away. When the documents are sent to Elasticsearch, they are further processed by the Elasticsearch client depending on the mapping used. The markdown text for instance was processed with the built in analyser that divides text into terms on word boundaries, using the Unicode Text Segmentation algorithm. It removes most punctuation, lowercases terms and removes stop words. After the markdown documents were uploaded to Elasticsearch, it took around 15 minutes for Elasticsearch to finish processing the text in the documents. After that process is done, all documents indexed were searchable using the Elasticsearch Python API.

```
es_df = tes_df

def gen(es_df):
    for i, row in es_df.iterrows():
        t = {
            "_id": row['nb_id'],
            "html_url": row['html_url'],
            'name': row['name'],
            'language': row['language']
            }
        yield t

errors = []
for ok, action in helpers.parallel_bulk(client=es, index="info", actions=gen(es_df
    if not ok:
        errors.append(action)
```

Figure 4.3: Code index fill method

4.1.3 SERP

The SERP or Search Engine Result Page is needed to show results of search queries. To achieve this, a Python Django server with port forwarding is used. It sends requests from the elastic search indices and updates a html page with the results of that search. It calls a Python file when the search is executed that searches the indices and returns the result. It also includes the query classifier and ranking functions. To create the basis of the SERP, code from Dritto 2018 was used to be able to use Elasticsearch with Django, since Django servers typically use an SQL server which was not desired here. The result page includes a searchbar, a language setting and an index setting. When results are found, up to twenty results are returned. It spawns a html div container that contains a clickable link with the name of the result and available metadata. Figure 4.4 below shows the response to the query "titanic".



20 results were found

titanic.ipynb
I've seen this dataset and worked with it before in terms of predicting survival in Udacity's [Intro to Data Science] (https://www.udacity.com/course/intro-to-d...
language: python

Count titanic survivors.ipynb
Let's count the number of women that survivedHow to merge the two loops into one?...
language: python

who_survived_the_titanic.ipynb
Who Survived the Titanic?Author: Notebook was contributed to the GraphLab team by William Lucia, who was a student at University of Insubria, ...
language: unknown

titanic.ipynb
Titanic Survival Analysis By Andrew Gill I am in interested in what factors contributed to survival on the Titanic and if i can reliably predict if a passenger will survive? (> 80% success rate)....
language: python

kaggle_titanic_script.ipynb
Read the training data raw = pd.read_table('../input/train.csv', delimiter=',')# Check which columns have missing values raw.apply(lambda x: sum(x.isnull())/len(x))# Look at estimating the missing ...
language: python

Figure 4.4: Example results page

The text snippit below the name of the notebook file in the figure is looked up from other indices when available. It searches the indices until a suitable snippit can be produced. When there are multiple options, the order of the availability is the following: markdown, comments and lastly parts of code. The language of the notebook is also included at the bottom of the html div container.

4.1.4 Removal of indexed empty and duplicate files

After using the SERP it was clear that empty and duplicate files needed to be ejected from the indices. The removal of the empty files is done by checking for each indexed document in the information index if that entry has no code, markdown and comments. If this is the case for a particular document, that entry is removed from the info index. Because all files needed to be tested, another Elasticsearch plugin needed to be used, namely the scan plugin, which gives a unique id to a chunks of documents in the index and iterative goes trough these chunks to be able to eventually go trough all indexed files.

For the removal of the duplicate files a more advanced solution had to be produced. The scan plugin is used anew to convert every indexed file to a hash value to compare them. This method is also described on the website of Elasticsearch by Marquardt 2020. The indexed file are transformed into a tuple with their markdown and the imports, classes and functions from the code that uniquely identify a specific document and are turned into a single hash value that is put into a Python dictionary. This is more memory efficient because only the hash values need to be stored in memory at the same time. The associated value of each dictionary entry is be an array of the index ids of the documents that map to the same hash value. If more than one document has the same hash value, then only one of the documents need to be retained. The rest can be deleted. A document was only deleted from the information index after a second examination round to affirm that all documents that map to the value are identical. This also means that documents that are almost the same will still be in the indexed documents.

4.2 Ranking

The first version of the search engine used simply the title to search through the information index. This is done using a query_string query. This query combines the search results of every word in the query while ignoring invalid parts of the query string. This simple method was then expanded upon to use a combination of the other indices. For this, a ranking method is needed to change the Elasticsearch scores to combine the different retrieval scores of the indices. There are 3 different indices that require a weight to be combined to rank their returned documents based on a query. These indices are the code, comments2 and markdown. A higher weight could be given the code or markdown index when a query was about code or text. For that purpose a classifier can be built. Using that code/text classifier the scores that are returned by Elasticsearch for each index can be boosted based on a vector with weights.

4.2.1 Classifier

A classifier is needed which needs to discern between source code and text (markdown). To achieve this, non empty samples were collected from the code index and the markdown index and were put in a dataframe with their corresponding label. The data is then split into a training and test set of 10290 and 3430 samples respectably. The scikit learn TF-IDFvectoriser is used to transform the markdown or code into a feature vector containing the term frequency–inverse document frequency features of the query that the logistic regression algorithm can use to classify with. The classifier has a 96% mean accuracy, as shown in further detail in figure 5.2. To tune the algorithm the different available hyper parameters of the logistic regression classifier they were compared using GridsearchCV of the scikit learn package. Gridsearch tries every parameter specified to include of the chosen algorithm and uses 5 fold cross validation until it exhaustively tried all combinations of tuning parameters. After executing, Gridsearch returns the algorithm with the highest cross validation accuracy score on the test set.

A random forest classifier that uses multiple decision trees was also tried by me but returned similar results after hyper parameter tuning.

4.2.2 Ranking queries

After building the classifier the next step is the weight and ranking function. This is done by fist running the query on all indices separately and recording the ids together with the score assigned to Elasticsearch. Then a weight vector is initialised as $[1, 1, 1]$, which is immediately normalised by summing the weights to one. The code/text classifier is then run on the query to classify it as either code or markdown. A higher weight is then assigned to the according index based on the result of the classification. If the outcome of the algorithm is "code" for example, the code weight gets boosted by one. After this, the weight vector is normalised to sum to one again. The weights then get applied to the Elasticsearch scores for each index. linear combination that sums the weight of the three indices to one.

4.2.3 Evaluation queries

To be able to compare search results, information needs and queries were needed. Because there was no dataset found for this purpose some information needs including corresponding queries were written in table 4.2 below. These queries will be used to input into the search engine to acquire relevance results.

Information need	Query
I want to know how to use the numpy function called unique by example	numpy unique
I want different examples of what genetic algorithms are used in Python?	genetic algorithms
I want machine learning examples with the IRIS dataset	Machine learning IRIS dataset
I want a sorting algorithm optimized for short lists	Short list sorting algorithm
I want an algorithm that can divide language into such a tree	Language tree parser
I want to separate my pictures from cats and dogs	Animal classifier
I want a neural network explaining the parts	Neural Network
I want an algorithm that can generate movie scripts	Movie script generation
I want the answers for the course Leren assignments	Leren regression neural network iris dataset sudoku
I want pictures of spiderman	pictures of Spiderman

Table 4.2: Information needs with corresponding queries

Chapter 5

Results/Evaluation

5.1 Classifier

The stock logistic regression classifier from the scikit learn package returned a mean accuracy of 96 percent on the test set as shown in figure 5.1 below. After using the hyper parameter tuning that was only increased by 0.003.

```
             precision    recall  f1-score   support

       code       0.97      0.95      0.96      1705
   markdown       0.95      0.97      0.96      1725

avg / total       0.96      0.96      0.96      3430

mean accuracy = 0.9603498542274053
```

Figure 5.1: Base Logistic Regression Classifier

After tuning the hyper-parameters of the logistic regression classifier using gridsearchCV, a small improvement could be made compared to the standard parameters. This is shown in figure 5.2 below. These improvements consist of a 0.3 percent improvement on the mean accuracy score and a one percent improvement on the code recall and the markdown precision metric.

```
             precision    recall  f1-score   support

       code       0.97      0.96      0.96      1705
   markdown       0.96      0.97      0.96      1725

avg / total       0.96      0.96      0.96      3430

mean accuracy = 0.9635568513119533
```

Figure 5.2: Tuned Logistic Regression Classifier

A more advanced RandomForest classifier was also tested, but had the exact score of the tuned Logistic Regression so no further improvements were made.

5.2 Evaluation of Queries

The queries in figure 4.2 at the end of the last chapter were put into the search field on the webserver. The top 10 results were then classified as relevant or not relevant. The results are displayed in table 5.1. Searching with only the title takes less than a second, searching all indices can take up to 10 seconds depending on the query used and current occupation of the hard drive. Some notebooks were no longer available. The relevance of these notebooks was thus set to Non Relevant. This would explain some of scores.

Query	Relevance title	Relevance all indices Classifier
numpy unique	[N,N,N,Nd,N,N,N,N,N,N]	[R,R,R,R,Nd,N,R,R,R,R]
genetic algorithms	[N,N,N,Nd,N,N,Nd,N,N,N]	[R,R,N,R,R,Nd,R,Nd,R,R]
Machine learning IRIS dataset	[R,N,N,Nd,N,N,N,N,N,Nd]	[R,N,N,R,N,N,R,N,N,Nd]
Short list sorting algorithm	[N,N,N,N,R,N,N,Nd,R,R]	[N,N,R,R,R,R,N,N,Nd,N]
Language tree parser	[N,N,Nd,N,N,N,N,N,R,Nd]	[N,N,N,N,R,N,N,N,R,N]
Animal classifier	[N,N,N,N,N,N,N,N,N,N]	[Nd,Nd,Nd,N,N,N,Nd,R,R,Nd]
Neural Network	[N,N,Nd,N,N,N,N,N,N,N]	[N,N,R,N,R,N,Nd,R,R,R]
Movie script generation	[N,N,N,N,N,N,N,Nd,N,N]	[N,N,N,N,N,N,N,N,N,N]
Leren regression neural network iris dataset sudoku	[Nd,N,R,R,N,R,R,R,R,R]	[R,N,N,N,N,R,Nd,R,R,N]
pictures of Spiderman	[]	[N,R,N,N,N,N,N,N,N,N]

Table 5.1: Relevance of the top 10 search engine results (R for Relevant, N for Non-Relevant Nd for not available files)

The title was mainly used to test single word queries, it struggles with multiple query words that were used for 5.1. The relevance of the search engine that used all indices is improved. When the classifier was not used to boost the weights, the query "numpy unique" got one more wrong: [R,R,R,N,Nd,N,R,R,R,R].

Chapter 6

Conclusion and Discussion

In conclusion the separation of the different notebooks cells had a positive influence on the relevance scores. When only using the information index, the document contents were not used and thus the evaluation score was lower. When the code/text classifier was not used, it could create less relevant documents as seen with the query: numpy unique. The 230 thousand duplicate files that were removed answers question two about the duplicate documents from the introduction. The n Does removing notebook files change the outcomes of the research by Rule? What queries and information needs will be used to to evaluate the different indices?

Removing the 450 thousand notebook files did not influence the outcomes of the research of Adam rule. The effect was minimal on the tested variables.

A few complications were encountered during this project that were important to mention. The data handling and preprocessing for this project used the biggest portion of the allocated project time. A difficult part of the ambitious to write own queries and because unlike Grappiolo et al. 2019, who had two experts classify 84 documents as relevant or non relevant, the relevance checks needed to be done by one person. This should be atleast two people so that metrics like cohens cappa score can be used. There are also still near duplicate files present in the index because only exact duplicates were removed from the index. These can on differ by a few lines of code. For some searches it is therefore possible taht alot of near duplicates are returned together like was the case before the removal of the exact duplicate files.

Chapter 7

Future work

Further improvements to the search engine can be made by adding a scraper that searches Github for newly added notebooks like the SSC of Grappiolo et al. 2019 used. This would make the search engine be able to search newly added document by indexing them immediately when created. The result of this feature would furthermore vastly increase the amount of data available to train and test models on and might enable the use of big data and with that the use of more advanced algorithms than the feature vector and logistic regression combination used in this paper. The use of a deep neural network for example would benefit from the increase inflation of data.

The second part to be improved would be the query preprocessing. Elasticsearch handles this for this project, but a custom query preprocesser could be used as used in Grappiolo et al. 2019. Furthermore the amount of near duplicate documents could be investigated.

Thirdly being able to choose every individual indices to use to produce better comparisons in relevance between all of the separate index to form a more informed conclusion.

There could also be made improvements to the code/text classifier. This could be done by tuning the hyper parameters of other algorithms other then Logistic regression and the Random forest classifier to achieve a higher mean accuracy score on the test set. The amount of boosting the indexes get from the classifier can also be tuned in the future with a factor in the project code.

The last improvement could be to the Ranking. The way it is implemented could be improved from the way it is done in this project. This could be done by customising the scoring function used by Elasticsearch to compare the query directly to the vector space model using cosine similarity. This is still an experimental feature of Elasticsearch and requires your indices to have vector fields to easily compute the cosine similarity with the query vector. The similarities could then be used in the scoring of the indexed Jupyter notebooks. Because the indices in this project did not have vectors in its mapping yet a more straightforward approach was used for this project. Multiple people should also be able to give their relevance judgements of different queries to calculate metrics like cohens cappa score.

Chapter 8

Appendix

The code used for this thesis is found on Github: here

Bibliography

[BK16] Christian Buck and Philipp Koehn. "Quick and reliable document alignment via tf/idf-weighted cosine distance". In: *Proceedings of the First Conference on Machine Translation: Volume 2, Shared Task Papers*. 2016, pp. 672–678.

[Dri18] Giovanni Pagano Dritto. *searchsite*. https://github.com/thalderg/searchsite. 2018.

[Gra+19] Corrado Grappiolo et al. "The Semantic Snake Charmer Search Engine: A Tool to Facilitate Data Science in High-tech Industry Domains". In: Mar. 2019, pp. 355–359. DOI: 10.1145/3295750.3298915.

[Lan19] Jenna Landy. *notebook-research*. https://github.com/jupyter-resources/notebook-research/tree/master/analysis_notebooks. 2019.

[Mar20] Alexander Marquardt. *deduplicate-elasticsearch*. https://github.com/alexander-marquardt/deduplicate-elasticsearch/blob/master/deduplicate-elaticsearch.py. 2020.

[Rul18] Aurélien; Hollan James D Rule Adam; Tabard. "Exploration and Explanation in Computational Notebooks." In: *UC San Diego Library Digital Collections*. 2018. DOI: 10.6075/J0JW8C39.