Creating a Jupyter Notebook "cell-type aware" Search Engine (*JuNocta*)

UNIVERSITEIT VAN AMSTERDAM

Kennet Botan
11058900

Bachelor Information Studies
Faculty of Science
University of Amsterdam

Supervisor
Dr. M.J. Marx
maartenmarx@uva.nl

05-07-2022

Abstract

Jupyter Notebook is commonly used and liked as computational notebooks. The notebooks can form an interactive way to explore data or give explanation on topics. The possibility to combine code, computational output, explanatory text and visualisations (and multimedia) are at the root of the popularity. However, at the time of writing this thesis there is not a way to search within a collection of these notebook files while (maximising the) understanding the json format of raw jupyter notebook files. A search engine, called JuNocta (Jupyter Notebook cell-type aware), is created that can (download a dataset) index that dataset, set up elasticsearch locally and run the search engine automatically/in one run. The results of the query are linked to the cells or files on the localhost/8888 or GitHub. A user study, in which questions are asked to gather 10 information needs within the collection of notebooks using two different search engines, is made in order to look at the possible advantages and disadvantages of both. The first search engine indexes the notebooks as text and provides no advanced query options, while the other was JuNocta and has advanced query options. Overall testing proved a significant difference in precision, as the search results of JuNocta had a higher chance of being relevant. This was achieved by the possibility to split a query, match exact phrases, specifying fields, cell types, code output types and indicating whether to search cell- or file-based.

Acknowledgements

I would like to express my gratitude to my supervisor dr. Maarten Marx. To understand the situation I was in and giving me multiple second chances to restart the project. To provide guidance to finally bring it to a successful end. I would also like to thank my sister and girlfriend for supporting me.

Table of contents

1. Introduction

Even if you are not in the world of IT, you might have heard of the programming language Python as it has become one of the most popular programming languages. Chances are less likely, you have heard about Jupyter Notebooks. Jupyter Notebooks provide a web-based interacting platform to run code, this is opposed to writing code in traditional text editors. It provides the possibility to use over 40 coding languages, but its default code language is Python. Make no mistake, this platform runs not on the internet, the code runs on your computer and is only displayed in a browser, like Chrome. On the webpage, the notebook is divided into cells that are displayed below each other and the user can indicate for each cell whether it runs code, formatted text, unformatted text or a heading. Traditionally, when a programmer wants to test his code, he has to run the whole file containing all the code. In a notebook however, each cell can run separately.

When saving a notebook, not only the text/code within the cells is saved but all the characteristics of each cell and some characteristics of the file itself are saved as well. Some of that information is useful, while some of it is abundant. The notebook will be saved as an IPYNB file that has the same structure of a JSON file. JSON is a standard file format that is text-based, and represents structured data in the form of attribute-value pairs and arrays. The notebook can be saved and stored on a computer, but if one wants to share his code with others, he can share it on GitHub.

GitHub is an open-source platform on which, among other things, developers can host code and work together on projects. GitHub can be considered one of the most popular platforms as 84% of the Fortune 100 companies use their software (Friedman, 2021). Mid 2022 there are over 14 million Jupyter notebook files hosted on GitHub. GitHub provides an advanced search, however the filters provided are limited as it is not designed to search for or within Jupyter notebooks. When searching for notebooks that are stored locally on your computer, the search bar in the file explorer on your computer faces the same issue. At the time of writing this thesis there is not a way to search within a collection of these notebook files while maximising the understanding of the JSON format. This means if someone wants to search for a notebook file, he can basically only search on the filename of that notebook and not in its content. This is a missed opportunity since there is a lot of unused information within this JSON structure that might have great potential. This paper tackles that by creating a search engine that will solve the previously described problem.

The search engine developed in this paper will search for notebook files or for cells within a notebook, both based on its contents. Such a search engine could be very helpful for a programmer who has created a lot of Jupyter notebook files over the years and wants to search for a specific file on his computer, from which he remembers a few key points. Another example could be if one wants to know how to write a specific snippet of code. If someone wants to learn about a certain subject, he could clone a few GitHub repositories (that contain notebooks) that are explanatory about that subject and search within that collection. The search engine should be able to understand the notebook format and determine which data is important and which is not, when indexing the data. During the indexing it should be aware of the different cell types, therefore the search engine will be called *JuNocta* (Jupyter Notebook "cell-type aware"). *JuNocta* should understand the type of query, for example whether it should search for code, text or figures. Therefore, six attributes have been drawn up for the search engine to meet: three attributes related to understanding the JSON format and three attributes related to querying.

In terms of understanding the JSON format, *JuNocta* should contain the following three attributes: (1) be able to search for a specific cell or for an entire file, (2) make distinctions between the different cell types and (3) make distinctions between the different code output types. For example, if the user wants to search for a neatly constructed notebook that at least uses headings and figures to showcase the data. Using filters, it should be possible to search for or a file that at least consists of the cell_types *heading, markdown* and *code* and the output_types *stream*, *execute_result* and *display_data*.

In terms of querying, the JuNocta should contain the following three attributes: (1) be able to search for an exact string, (2) make use of Boolean operators and (3) to a certain extent make use of regular expressions. For example, if the user wants to search for a way to count the unique values within a pandas DataFrame. Assuming the user has prior knowledge of pandas, he will know that *pd.DataFrame* usually gets the variable name *df*, he could use this knowledge to his advantage when creating a query. The following query could be created in order to get the desired result: *(dataframe OR df*) AND unique AND value*.

The Search Engine Result Page (SERP) is the page in which the results of a query are displayed. The design should be neat with not too many functions to distract the user and a big entry box for the user to enter the query. The results should be displayed similar to the SERP of Google, this design has proven itself through time and why change a winning formula.

While developing the search engine, the size of the dataset will be increased in three steps. The three datasets that are being used in chronological order of size are: the notebooks of the books by Jake VanderPlas: *Data Science Handbook* and *A Whirlwind Tour of Python* (around 100 notebooks), the notebooks referred to by the wiki page on the GitHub repository *jupyter* of the user *jupyter* (around 3000 notebooks) and a dataset of notebooks collected by the researchers Rule A et al. (around 200.000 notebooks). The first and third dataset are downloaded. The second dataset is created by a script that web scrapes the wiki page and clones the repositories, in which the notebooks reside, that were referred to.

The main research question is: *In what ways does JuNocta improve searching for or within jupyter notebooks files compared to a search engine that only indexes the notebooks as text?* For each dataset the following sub questions are asked:

1. To what extent is it possible to run a search engine on your own laptop, that is designed for Jupyter notebooks and with just one push of a button can download, index entire sets of GitHub repositories and make the notebooks locally searchable?
    1. With what success rate can the repositories mentioned on the wiki page be cloned on/to the user's computer (this question is only relevant to dataset 2)?
    2. With what success rate can the notebook files be indexed into a pandas DataFrame?
    3. With what success rate can the notebook files (pandas DataFrame) be inserted to Elasticsearch on the localhost/9200?
2. Is an advanced cell-type aware search engine preferable above a search engine that only indexes the notebooks as text?
    1. What is the precision of each search engine?
    2. What are the differences between using the two search engines in order to gather the information needs for each dataset, what are the possible advantages or disadvantages?

2. Background

The process of using a search engine starts consecutively with the existence of a SERP, a user that types a query to search, the search engine determining which documents are most relevant for this query, the documents being displayed in the SERP and ready for the user to interact with. One of the most important parts of a search engine is determining the relevance of documents for a certain query. By means of scoring each document a ranking can be established. In this project, JuNocta lets Elasticsearch score the document. In this chapter Elasticsearch is covered.

2.1 Elasticsearch

Elasticsearch is an open-source search and analytics engine for all types of data. According to StackShare, a platform where developers and companies rank software, Elasticsearch is reportedly used by 3730 companies like Uber, Shopify and Slack. Since January 2016 Elasticsearch has been the most popular search engine software according to the DB-Engines ranking. Popularity is an important element to programming languages or software as it tends to have a bigger community. This larger community provides more forums in which questions are asked and problems are solved. Therefore, unique problems are more likely to be answered. The same applies to the existence of supporting libraries. In addition, I have prior experience with Elasticsearch from a course called *Search Engines*. In this course the students needed to create a search engine on a smaller scale. I enjoyed this course, this added to my decision to make it the subject for this thesis. Above points have helped my choice to use Elasticsearch for this project.

In order for JuNocta to work, there needs to be a collection of documents to search within. Simply put, there are two ways to connect with this collection: one that is stored on the internet and one that is stored locally on the computer itself. JuNocta uses Elasticsearch to run locally on the user's computer. When given a query, Elasticsearch will use their default BM25 algorithm in order to rank the top results.

3. Related work

In this chapter, scientific papers relevant to the search engine topic are discussed. These papers can be categorised into: papers about Jupyter Notebooks and papers about search engines used to search code.

Perkel (2018) describes the rise of the Jupyter Notebook and why it is commonly used and liked by data scientists as computational notebooks. The notebooks can form an interactive way to explore data or give explanation on topics. The possibility to combine code, computational output, explanatory text and visualisations (and multimedia) are at the root of the popularity. These options make it viable to use notebooks in a supporting role to scientific papers (Kluyver et al., 2016). Accessibility is not an issue either, since it is not mandatory for the notebooks to run on the user’s computer. Code could be run on a supercomputer provided by an institution or notebooks could entirely run in the cloud. However, the easy accessibility might form an issue if there is not a uniform style in which a notebook should be constructed by its users. Pimentel et al. (2019) did a study on 1.4 million notebooks collected from GitHub and looked at good and bad practices by the users. These bad practices include out-of-order cells, non-executed cells and possible hidden states. In Rule A et al. (2018) the relation between exploration and explanation in these computational notebooks is explored. Three studies were conducted, in the first study an analysis of 1.25/roughly 1 million Jupyter Notebooks found on GitHub is made. Their conclusion was that about a quarter of those notebooks contained no text, but consisted only of code and visualisations. The two other studies provided more evidence for this precedent. These non-explanatory notebooks could be considered personal by its owner and therefore be messy. The study does give their view of how a notebook should look like. According to them a notebook should start with the notebook title and introduction (text) and continue with the importing of external packages (code), description of model parameters (text), implementation of parameters (code), description of need to profile data (text), profile plotting code (code), inline plot (visualisation). In Rule A et al. (2019) researchers delve deeper into how a computational notebook should look like and have written ten simple rules to follow. When a notebook follows this guideline, we can consider this notebook ‘bookshelf worthy’.

Since there are no scientific papers about search engines specifically designed to work with jupyter notebook files, we will look at papers about search engines for other programming languages or file formats. Despite not being exactly the same, these search engines might still be similar and useful.
In Chatterjee S et al. (2009), a search engine called SNIFF (SNIppet for Free-Form queries) was designed specifically for java using free-form queries. Free-form queries are queries that are formulated in a natural language (for example plain English), meaning that the user needs no prior knowledge about an information need. The results of the queries were code snippets. Despite the difference between java and jupyter notebook, there is a resemblance between this search engine and *JuNocta*, since the java code snippets can be compared to code cells in jupyter notebook (not looking at the actual language itself).

In Imminni S et al. (2016), a search engine called SPYSE (Semantic PYthon Search Engine) was designed specifically for Python packages and modules. The packages are JSON files containing its metadata like the author, description, package_url etc. The modules are Python files, containing the classes, methods and variables. Elasticsearch was used to index their data and query the indexed data. SPYSE shows a lot of resemblance with *JuNocta*, since both make use of Elasticsearch and both the Python packages and Jupyter Notebooks are written in JSON, containing metadata. The Jupyter notebooks contain at the top-level: *metadata*, *nbformat*, *nbformat_minor* and *cells*. However, the focus in their search engine was on the metadata, while in *JuNocta* the focus will be, as the name suggests, on the *cells*. Besides, SPYSE makes use of weighted zone scoring, as it gives different weights to zones. For example, it gives more relevance to the *class* field, as opposed to the *function* field.
In Kim K et al. (2018) a search engine, called FaCoY (Find a Code other than Yours), was designed that uses a code-to-code approach, as opposed to the commonly used free-form approach. This approach means that the user will put code as input and will get semantically similar code fragments as result. The code-to-code approach did already exist, however the FaCoY has a slight difference. Their code-to-code approach does not try to directly match the code from the query with the code in the collection. Instead, firstly it tries to look for tokens, within the given query, that show similar functional behaviour. They call this *query alternation* and it proves to be more successful in finding code fragments implementing similar functionalities.

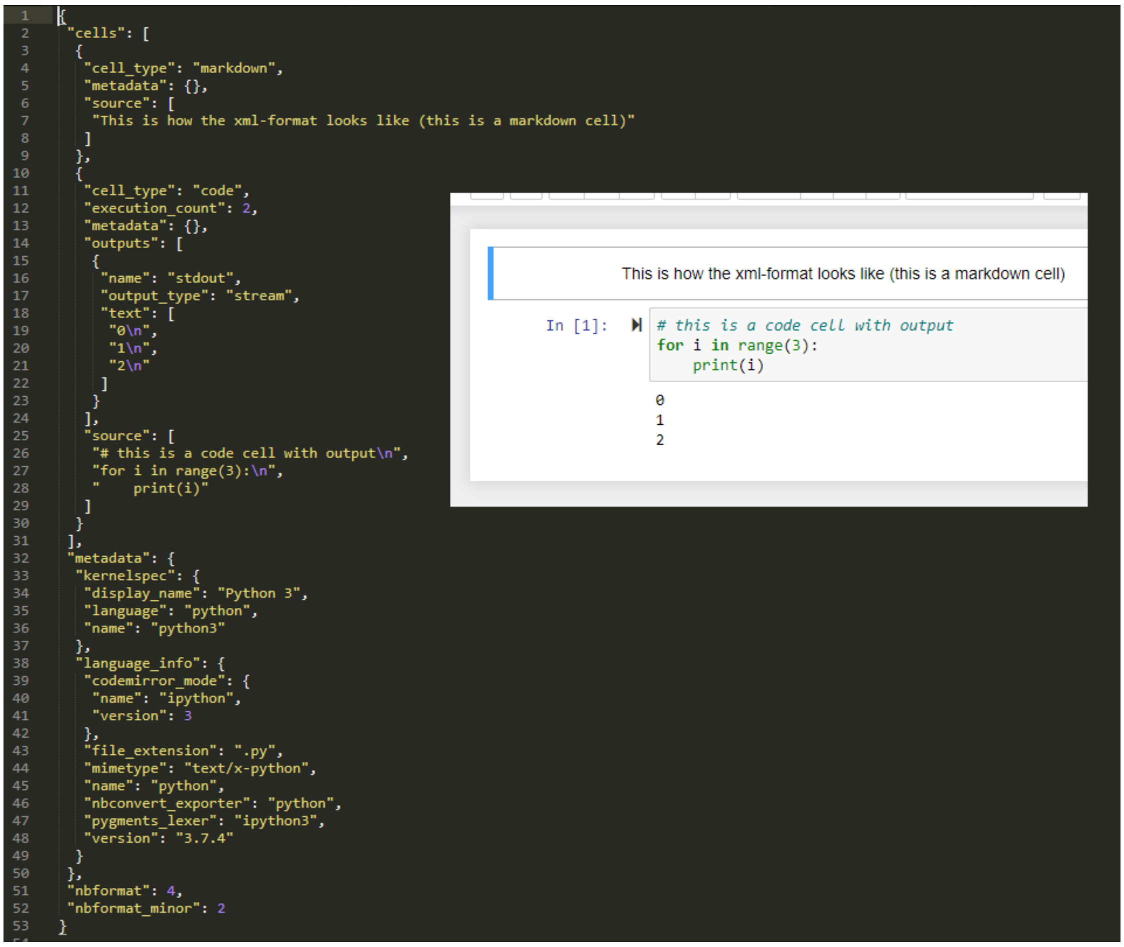
4. Methodology

In this chapter the methods used to develop *JuNocta* are outlined. This will be consecutively done by describing the data that will be used, how the data is gathered, how the data is indexed, how the indexed data is stored locally in Elasticsearch, the development of the SERP, the creation of queries and the interaction between the GUI and SERP. The chapter ends with a user case study with ten information needs to test *JuNocta*.

4.1 Description of the data

To understand how the process of creating a search engine works, first an understanding of the Jupyter Notebooks files must be established. As described in chapter *1. Introduction*, JSON is a file format that is text-based and represents structured data in the form of attribute-value pairs and arrays. Jupyter Notebook files are structured in JSON-format (see *figure 10*). The structure of every Notebook file starts at the top level with an array of four items containing: *metadata*, *nbformat*, *nbformat_minor* and *cells*. The first item is *metadata*, this is a dictionary that describes the data used in the file. This dictionary consists of *kernelspec*, *language_info*, *file_extension* etc. Given that the notebook format has evolved over the years, changes have been made to the format and thereby multiple versions have been developed. The second item is the *nbformat* (notebook format), this is an integer that represents the version of the notebook format in which the notebook was constructed. The third item is the *nbformat-minor*, this is also an integer and represents whether backward-compatible or backward-incompatible changes of the notebook format are made. The last item of the array is the *cells*, this is a list of dictionaries, where each cell in the notebook file has its own dictionary containing data about the corresponding cell.

In the figure below, a notebook file (containing two cells) in two different forms is displayed. On the left side the raw json form is shown and on the right side the browser-based graphical interface of Jupyter notebook is shown. The figure should provide visual support for the understanding of the structure that has just been described.



```
{
 "cells": [
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "This is how the xml-format looks like (this is a markdown cell)"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 2,
   "metadata": {},
   "outputs": [
    {
     "name": "stdout",
     "output_type": "stream",
     "text": [
      "0\n",
      "1\n",
      "2\n"
     ]
    }
   ],
   "source": [
    "# this is a code cell with output\n",
    "for i in range(3):\n",
    "    print(i)"
   ]
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.7.4"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 2
}
```

This is how the xml-format looks like (this is a markdown cell)

```
In [1]:  # this is a code cell with output
         for i in range(3):
             print(i)

         0
         1
         2
```

Figure 10: The ipynb-file in JSON-format vs the two corresponding cells in jupyter notebook

For *Junocta* the focus will primarily be on the fourth item: *cells*. However, an exception lies with (the second item) *nbformat*, since the structure within *cells* depends on the nbformat version. For example *data['worksheets'][0]['cells']* in (nbformat) version 3 has changed to *data['cells']* in (nbformat) version 4. These differences in the (json) structure between different nbformat versions needed extra coding in order for the search engine to work universally.

An understanding of the file format and its relevant parts has been established, but how does a search engine fare with notebooks that differ? To see if *JuNocta* works accordingly in different environments, the search engine will work with these datasets. These datasets differ in tidiness and consecutively get bigger in size, roughly 100 to 3.000 to 200.000 notebooks. Whereas the first dataset is the tidiest, the third dataset is the least tidy.

4.1.1 Dataset 1: Jake Vanderplas notebooks

Jake Vanderplas is software engineer and has written the books *A Whirlwind Tour of Python* (98 pages) and *Python Data Science Handbook* (548 pages). These books explain how to use Python for certain situations, using text and snippets of code to support the explanation. These snippets of code are created in Jupyter Notebooks and Jake Vanderplas has published those notebooks publicly on his GitHub page. Each book has its own GitHub repository and contains Jupyter Notebooks. These notebooks are constructed very neatly, using a lot of explanatory markdown cells. The two repositories combined contain 86 notebooks, which leads to a total of around 4000 cells.

Moreover, Github uses a system in which users have multiple actions concerning repositories. The user can clone a repository, he can rate the repository by giving it a star, he can create issues, etc. The GitHub page *Python Data Science Handbook* has 31000 stars, making it the 216ste most starred repository on github. One can assume this repository is 'book-shelf worthy', I only give this recommendation if a collection only consist of neatly constructed notebooks. The repository for the book *A Whirlwind Tour of Python* is of similar style although smaller in size. This dataset 1 is considered the tidiest of the three.

4.1.2 Dataset 2: Gallery interesting notebooks

In *Chapter 3: Related Works*, the research of Rule et al. (2016) and Rule et al. (2018) was discussed. Both papers give a guideline in order to create a 'good' notebook. Their standard I called 'book-shelf worthy'. As Jupyter has its own GitHub page, it has published a gallery of interesting Jupyter Notebooks on the wiki tab of their repository *jupyter/jupyter*. By web scraping that wiki page, a collection of urls of GitHub repositories was created. All those repositories are then cloned to the user's computer and this is considered dataset 2, which contains 3117 notebooks (approx. 100.000 cells). The process of web scraping and cloning dataset 2 is described in chapter *4.2 Gathering dataset 2*. Cloning whole repositories, containing all kinds of file types, as opposed to downloading thousands of notebooks files into a single folder is comparable to a user setting his own directory as the dataset for the search engine to work with.

4.1.3 Dataset 3: EECN notebooks

The EECN dataset is part of a bigger collection of notebooks created by the paper *Exploration and Explanation in Computational Notebooks* (Rule A et al., 2018), henceforth the abbreviation EECN. The paper divided its collection, containing 1.25 million notebooks (found on GitHub), into six zip files totalling 270 GB. Dataset 3 is the first of six zip files called *Notebook files - part 1*, containing 198.778 notebooks (approx. 5 million cells) and has unzipped a size of 80GB. The researchers were able to fit all those notebooks into six zip files as a result of storage optimisation. However, this comes at the cost of excluding all external data files that a notebook might use. For example, a notebook works with a CSV file containing house prizes and creates a pandas DataFrame and graph. If the user wants to experiment with the code, rerunning the code is not possible, since the csv file is not present on the user's computer. Since these notebooks are scraped off public repositories on GitHub, there is no control over whether the owner follows certain guidelines to create tidy notebooks and therefore this dataset is considered the least tidy.

4.1.4 Short analysis of datasets (differences)

Since the notion of a 'book-shelf worthy' notebook is established, it can be concluded that this notion has a strong correlation with the ratio between the different cell types. To get an overview of the ratio between the different cell types and code output types, the following three figures are created: *ratio of cell types for all cells*, *ratio of output types for all code cells* and *ratio of files and their cell type composition*. Each figure shows the three datasets. The figures are made by first creating a pandas DataFrame, this process is described later on in *Chapter 4.3.1 Creating DataFrames*. The methods *groupby*, *count*, *size* and *transform* are used in order to create smaller DataFrames that display the data that are used for the three figures.

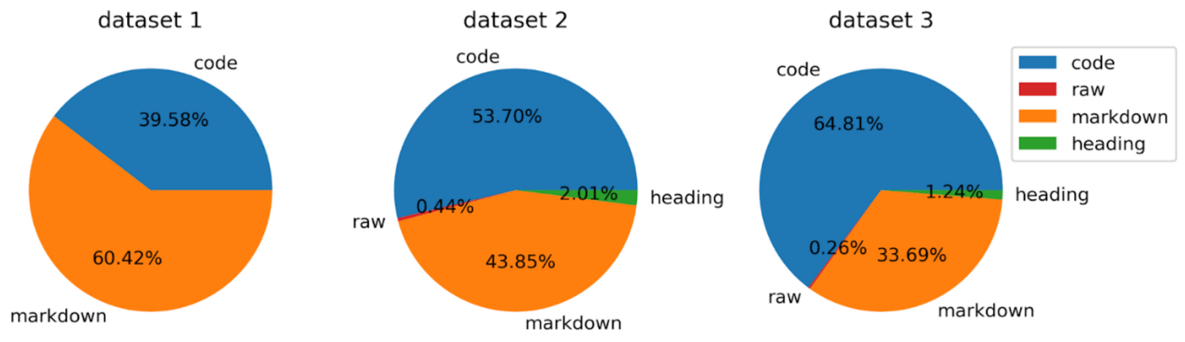


figure 11: Pie charts for each dataset, every chart contains each cell type found in the dataset and its share in all the cells in that dataset.

In Figure *11*, the pie charts show that dataset 1 has more markdown cells than code cells (60% vs 40%), this is expected since the code cells are there to support the explanatory nature of the notebook, which is mainly in markdown cells. Dataset 2 and 3 are the opposite of this, they have more code cells than markdown cells, while dataset 3 has relatively more code cells (65%) than dataset 2 (54%). This is expected given that Dataset 2 contains notebooks that are collected from repositories that are selected by *Juypter* worthy of praise and Dataset 3 contains notebooks that are scrapped from public GitHub repositories. Dataset 3 also included the cell_types *moarkdown* (1 cell) and *plaintext* (45 cells), these are excluded as I assume these are faulty typed or deprecated and are negligible in a dataset of 5.3 million cells.

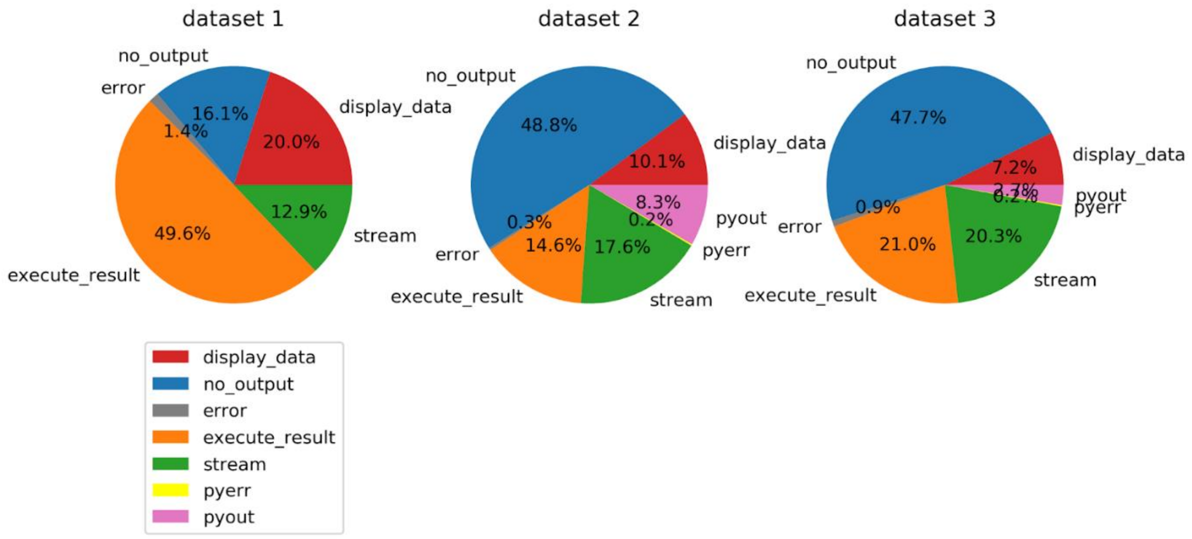


figure 12: Pie charts for each dataset, every chart contains each code output type found in the dataset and its share in all the code cells in that dataset.

In Figure *12*, the pie charts show that dataset 1 has relatively less code cells that are categorised with the label ‘no output’ compared to dataset 2 and 3. I assigned this label to a code cell if *cell[‘outputs’]* in the JSON structure of the notebook file for that cell contained an empty list. This is in contrast to a dictionary that would otherwise be in this list (see figure 10). For example, if only a function is defined within a cell and therefore the code will have no output. Dataset 2 and 3 are surprisingly similar, as I expected more difference. I expected this as there is a big difference in tidiness and assumed this would have some effect on the ratio of code output types.

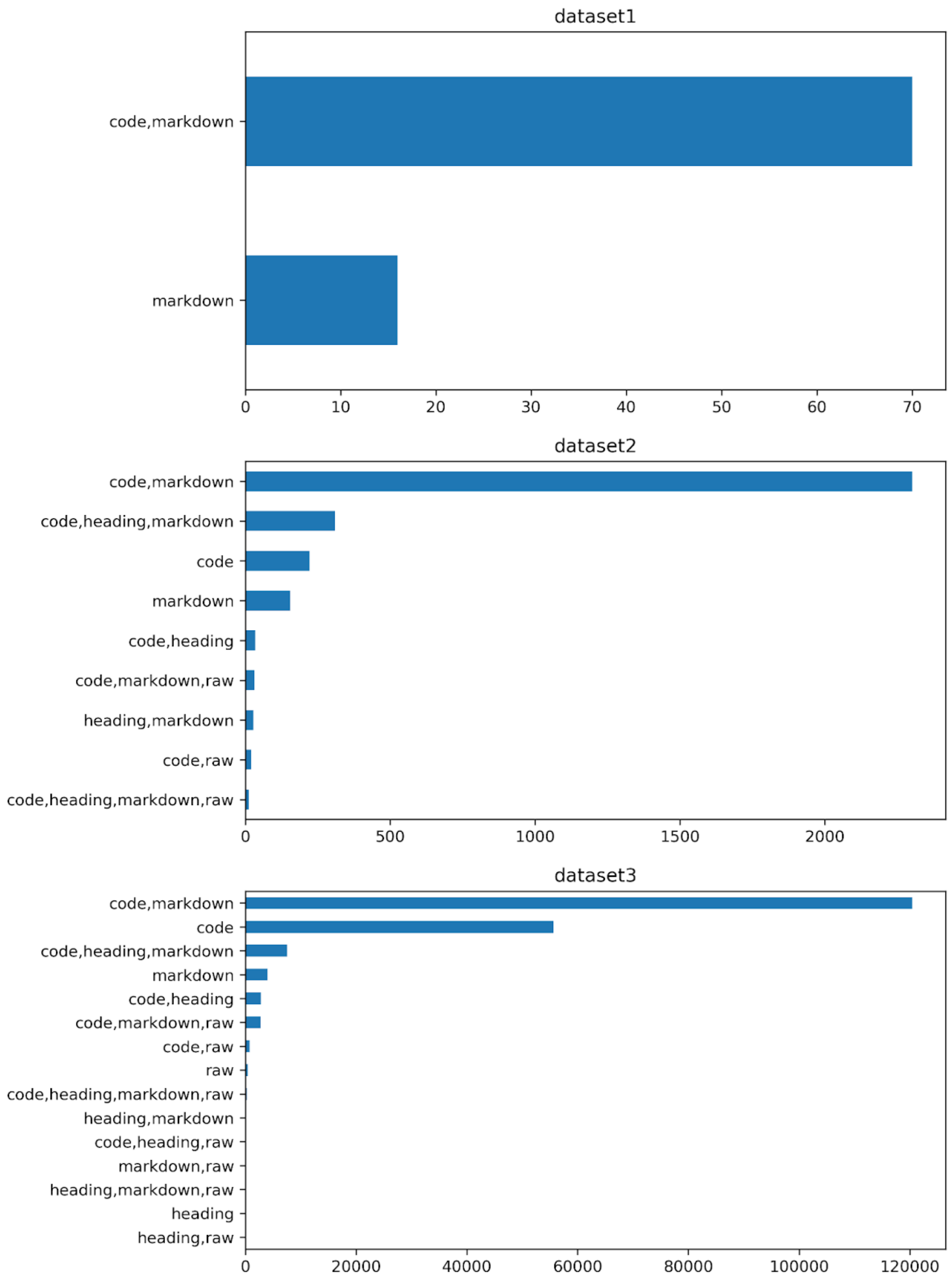


figure 13: Horizontal bar graph for each dataset, every graph contains the total occurrences of the compositions of cell types per notebook

In Figure *13*, the bar graphs show the composition of cell types per notebook of each dataset. Dataset 1 shows that a notebook either consisted of only markdown cells or both markdown and code cells. There are no notebooks that only contain code cells. As mentioned before, this is due to the explanatory nature of the notebooks in dataset 1. This is unlike dataset 3, where 29% of the notebooks only contain code cells. Overall, a decreasing trend in the combination *code,markdown* is present (81% > 74% > 62%).

4.2 Gathering dataset 2

Since both dataset 1 and 3 are manually downloaded, the focus in this chapter lies on dataset 2. This dataset has been created by web scraping a webpage for GitHub repositories, after which these repositories are cloned. Only the sub-chapters *4.2.1 Web Scraping*, *4.2.1.1 GitHub url pattern* and *4.2.2 Cloning* are purely focussed on dataset 2, the other sub-chapters in this chapter 4 are relevant to dataset 1 and 3 as well.

4.2.1 Web Scraping

The process of cloning all the notebook files associated with the page *A gallery of interesting Jupyter Notebooks* started by web scraping. Web scraping is the process of extracting/collecting data from websites. To understand web scraping, one must have a rudimentary grasp of how web pages are constructed. Every web page is constructed with source code, this is written in HTML language and it uses tags to indicate how to format and display the content. This source code is important, as it gives the whole layout and its content of a website while using different tags. For example, the <*a*> tag indicates a clickable link to another website. Web scraping extracts this source code, after which it can be analysed or worked with.

Web scraping can be done in a process extracting data from multiple websites in one go, using the same algorithm. In the case of this project, the focus lies on just one particular website (although it would work on other webpages as well). The website that needs to be scraped is the wiki page of the repository *jupyter* by the user *jupyter*. This wiki page consists of unordered lists of links to interesting github repositories.

For this project *BeautifulSoup* is used to scrape that web page. BeautifulSoup is a Python library that is used for pulling data out of HTML files. A list of all links (within the body) to websites is created. Four categories could be made out of those links: (1) a url that contained the string *github*, (2) a hash-link to a specific part within the wiki page, (3) a url of a nbviewer of which there was no way to extract a github repository and (4) a url of a different site. For example

- *https://github.com/jakevdp/PythonDataScienceHandbook*,
- *#data-driven-journalism*,
- *http://nbviewer.ipython.org/gist/3407544* or
- *http://lorenabarba.com*.

Of those four categories, only links that fall under the first category are continued with. In order to distinguish these categories, regular expressions are used. Six patterns are created in order to form a list of legitimate urls to GitHub repositories. The module *re* is used to distinguish those patterns. This module is a package built in Python, that provides regular expression matching operations. It will match patterns in all provided links to a GitHub repository url. If a link was a url to a specific ipynb file (in a subfolder) within a repository, the code would create a link to the repository itself by using regular expressions. This is done intentionally to clone more notebook files, since the goal was to create the biggest collection of 'bookshelf-worthy' notebooks and if one notebook within a repository is 'bookshelf-worthy', one can assume the rest of notebooks in the repository are as well.

4.2.1.1 GitHub url pattern

A GitHub repository url starts with a repetition of any possible character (except a newline) or has no prefix and is followed by (or starts with) '*github.com/*', followed by a '*username*', followed by a '*/*' and it ends with a *'repository'*. The characters of the *username* can only be a hyphen (-) or alphanumeric (lower- and higher case). The characters of the *repository* can only be a hyphen (-), an underscore (_), a period (.) or alphanumeric (lower- and higher case).

The GitHub url patterns could be divided into 3 patterns: (1) url of a repository, (2) url of a certain file within a repository and (3) url of a github user. Only the url of the GitHub user was not usable, since no reference to a repository could be made.

Although a set of urls that match the ‘github pattern’ is created, there is a possibility that one of those urls is a url to a page that does not exist anymore (see *figure 14*). On the page *A gallery of interesting Jupyter Notebooks* there are multiple non-existing pages. BeautifulSoup is used again to check the webpage and whether the repository actually exists. This is done by checking the <*title*> tag of the webpage. If the title is “*Page not found · GitHub*”, there will be no effort made to clone the repository.

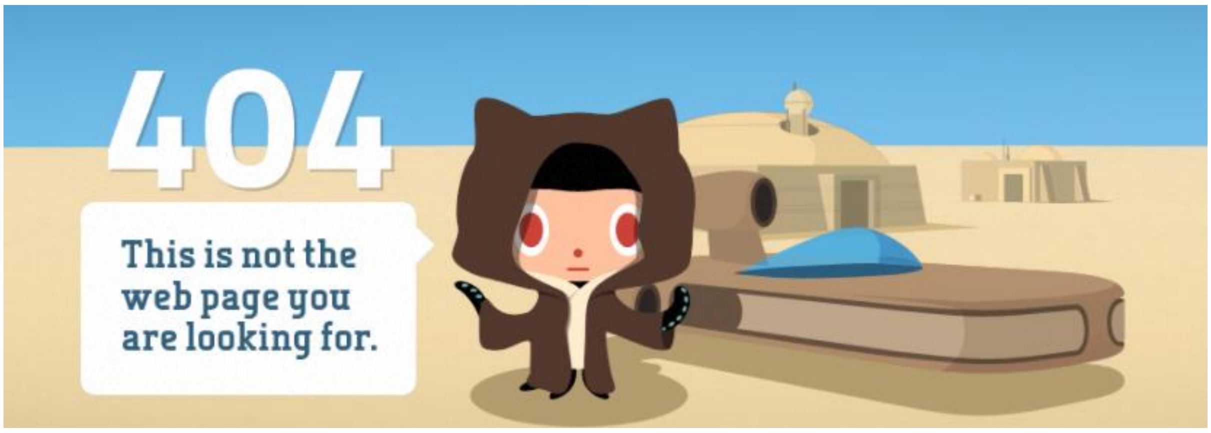


Figure 14: webpage that is shown for a url of a non existing repository

4.2.2 Cloning

When cloning the repositories to your own computer, the repositories are named in a specific format: *(user)repository*, for example *(jakevdp)PythonDataScienceHandbook*. The github username and repository are important, because together they form the url of the github repository. Since the name of the user and repository are not retrievable in the files itself, both are now in the folder name of the locally saved repository. With regular expressions these can be extracted in a later stage to create the url to the github repository. Programming-wise cloning the github repositories with a given url was straightforward with the *git* module. Without counting the lines *try* and *except* surrounding it, only one line of code was needed: *Repo.clone_from(url,repo_dir).*

4.3 Elasticsearch

Elasticsearch is an open-source search and analytics engine for all types of data. According to StackShare, a platform where developers and companies rank software, Elasticsearch is reportedly used by 3730 companies like Uber, Shopify and Slack. Since January 2016 Elasticsearch has been the most popular search engine software, to search within databases, according to the DB-Engines ranking. Popularity is an important element to programming languages or software as it tends to have a bigger community. This larger community provides more forums in which questions are asked and problems are solved. Therefore, unique problems are more likely to be answered. The same applies to the existence of supporting libraries. In addition, I have prior experience with Elasticsearch from a course called *Search Engines*. In this course the students needed to create a search engine on a smaller scale. I enjoyed this course, this added to my decision to make it the subject for this thesis. Above points have helped my choice to use Elasticsearch for this project.

Elasticsearch provides multiple ways to index data into Elasticsearch, it lets the user choose their own data architecture that fits his own needs. For example, by building a CSV file and indexing that data. I chose to build a pandas DataFrame. The DataFrame provides an insight into how the datasets look and gives a range of options to create visualisations for the datasets, these are shown in chapter *4.1.4 Short analysis of datasets*. After the DataFrame is created, it can be bulk loaded in on the localhost/9200. Since the search engine will work on cell and file based, two different Dataframes needed to be created.

4.3.1 Creating DataFrames

When the search engine is based on the cell, the following characteristics of a cell are saved: *cell_id*, *file_cell*, *file*, *nbformat*, *folder*, *user*, *repo*, *location*, *string*, *lines*, *cell_type*. The process for creating this Dataframe starts with understanding how many *.ipynb* files there are. The method *os.walk()* is used to create a dictionary with the file name as key and file location as value. Now we can loop through the .ipynb files, using this dictionary and its locations to open the file using the *open(..) as* .. and *json.load()* methods. As described in *4.1 Description of data*, the focus is on the dictionary *cells*, that is a list of cell dictionaries. Loop through *cells* and for each cell dictionary assign the characteristics, that are described at the beginning of this alinea, to a temporary dictionary. At each iteration this temporary dictionary is assigned as a value to the cell as a key in a dictionary that keeps all the cells of all the files.

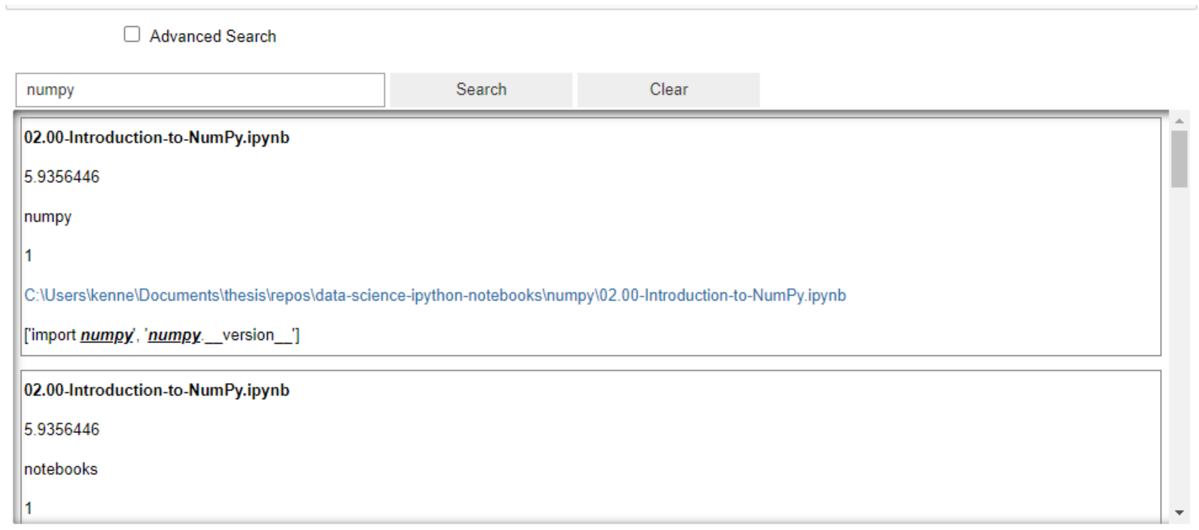
When the search engine is based on the file, the following characteristics of a notebook file are saved: *file_id*, *file*, *nbformat*, *folder*, *repo*, *location*, *string*, *lines*, *code_cells*, *code_lines*, *markdown_cells*, *markdown_lines*. The process for creating this DataFrame is similar to the process described above. The difference is that text of each of the cells are combined into a single string and the cell types and output types are stored as a 1 or 0 value for each type, whether the file contains that type or not.

4.3.2 Setup of Elasticsearch

Just as the cloning of the repositories, the process of setting up the Elasticsearch locally is fairly straightforward. The method *es.bulk()* is used to store the DataFrame into Elasticsearch locally. The more complex process regarding Elasticsearch is the creation of queries, which is explained in the chapter *4.4 Creating queries for Elasticsearch*. However, the *bulk* method occasionally does provide (some) difficulties, therefore the DataFrame was split into smaller chunks each containing 1000 rows and the *bulk* method was surrounded with a *try* and *except* statement. When an exception does occur, the *cell_id* of the first and last row of the chuck is known, after which further a possible investigation of why the exception can be held (with the aim of improving the success rate).

4.4 Development of the SERP

In order to have the user make use of the search engine, a graphical user interface (GUI) is needed. When I was in the exploration phase of testing how to handle user queries the standard Jupyter widgets (ipywidgets) were used, since I was already familiar with these widgets. Ipywidgets, also known as Jupyter-widgets or simply widgets, are interactive HTML widgets for Jupyter notebooks and the IPython kernel. In the figure below, the search engine operates in a (single) cell within a jupyter notebook file, while making use of the ipywidgets (see *figure 15*). From now on this ‘beta’ version of the search engine, that uses ipywidgets, will be called *JuNocta 0.9*.



```
02.00-Introduction-to-NumPy.ipynb
5.9356446
numpy
1
C:\Users\kenne\Documents\thesis\repos\data-science-ipython-notebooks\numpy\02.00-Introduction-to-NumPy.ipynb
['import numpy', 'numpy.__version__']

02.00-Introduction-to-NumPy.ipynb
5.9356446
notebooks
1
```

Figure 15: SERP JuNocta 0.9 using ipywidget

These widgets, used in *JuNocta 0.9*, were helpful and have a good use for simple tasks, however they are limited when working on projects with more complex layouts. A better collection of widgets was needed to cope with the more complex layout.

Besides choosing a better means of creating the GUI, another reason is the convenience for the end user who will use *JuNocta*. If a user wants to use *JuNocta 0.9*, the user must perform three tasks. (1) He must open the command prompt to open the jupyter notebook environment. (2) He must browse to the directory in which the notebook file is stored and open it. (3) He must run all the cells. If the notebook file is not split up into multiple files, each containing a different subject of coding, that file will be confusingly long for the user. However, if the file is split up, that will mean that the user has to open multiple notebook files and each time must run all cells of that specific file.
The new improved version, that will be called *JuNocta 1.0*, will have the user perform two simple tasks prior to the use of the search engine: open the command prompt and run two commands: *cd [path]* and *main.py*. The steps that the user will have to make in order to make use of *JuNocta 0.9* and *JuNocta 1.0* exclude the installation of the requirements. These requirements consist of Anaconda (Python and Jupyter Notebook), Java, GitBash and ElasticSearch.

4.3.1 Kivy vs Tkinter vs PyQT

In search of a new and more complex GUI, three options often came to the top of the search results and seemed most popular: Kivy, Tkinter and PyQT. After research of the three GUI, Kivy seemed best for creating mobile applications while it uses a special syntax for the kv language. PyQT is the most complex of the three and hardest to learn. PyQT has a QTDesigner with a drag and drop interface to design the layout of the application which can be seen as a benefit, however I did not want to use such an interface but design it manually myself by coding everything. The Tkinter library is Python’s standard GUI library, which means it is built-in and therefore, unlike Kivy and PyQT, you do not need to install anything. Besides that, Tkinter does not have its own special syntax, making it more accessible. When comparing the three options to each other, I deemed Tkinter the best for this project.

4.3.2 Tkinter: structure of application

Tkinter has three built-in geometry managers to control where widgets are placed in the user interface: the *pack*, *grid*, and *place* managers. The place manager places widgets in a two-dimensional grid using x and y absolute coordinates. The pack manager organises widgets in horizontal and vertical boxes. The grid manager places widgets in a two-dimensional grid using row and column absolute coordinates. To determine which manager would be best for this project, an early sketch for *JuNocta 1.0* was made. After experimenting with the different managers to replicate the layout of the sketch, the grid manager seemed best for this project and its preferred layout.

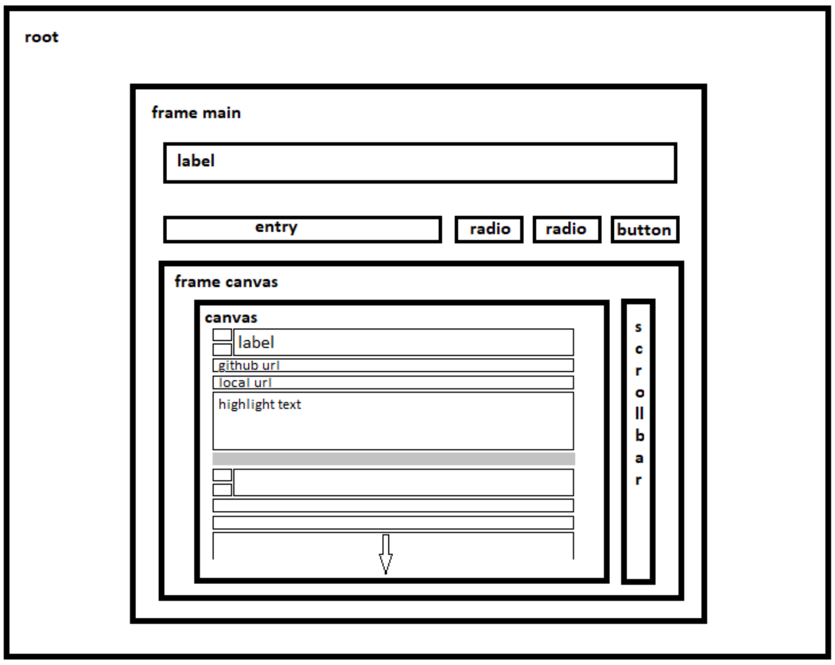


figure 16: early sketch SERP made in Paint

In order to create a sketch of how the layout will look like, first the Windows program *Paint* was used to get an idea (see *figure 16*). When the layout seemed to get more complex, the switch to the program *Pencil* was made. Pencil is a free and open-source GUI prototyping tool in which people can create mock-ups. The following detailed sketch was created (see *figure 17*).

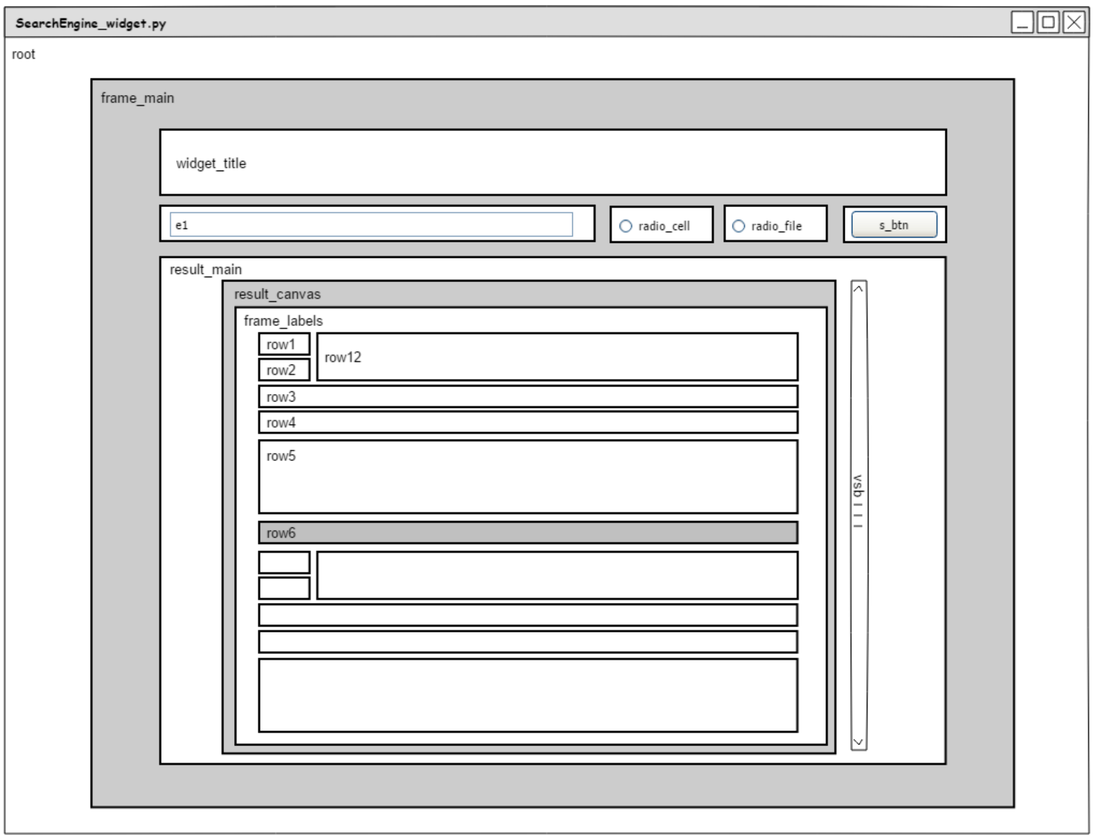


figure 17: detailed sketch SERP made in Pencil

The application was built by layers of widgets, while using the grid manager controls in which row and column each widget is placed. In total a minimal of 26 widgets were used, with a total of 11 unique widgets: Frame, Label, Entry, Button, RadioButton, MenuButton, Menu, CheckButton, Canvas, HTMLLabel, Scrollbar. The following structure of the GUI was created, each item in the unordered list below consists of (the level of the layer in which the widgets reside), the Tkinter widget followed by this widget's variable name in parentheses:

- 1 Frame (root)
    - 2 Frame (frame_main)
        - 3 Label (widget_title)
        - 3 Entry (e1)
        - 3 Button (s_btn)
        - 3 RadioButton (radio_cell)
        - 3 RadioButton (radio_file)
        - 3 MenuButton (m_btn_cell)
            - 4 Menu (m_btn_cell.menu)
                - 5 CheckButton + IntVar (cell_code)
                - 5 CheckButton + IntVar (cell_markdown)
                - 5 CheckButton + IntVar (cell_heading)
                - 5 CheckButton + IntVar (cell_raw)
        - 3 MenuButton (m_btn_output)
            - 4 Menu (m_btn_output.menu)
                - 5 CheckButton + IntVar (stream_v)
                - 5 CheckButton + IntVar (execute_v)
                - 5 CheckButton + IntVar (display_data_v)
                - 5 CheckButton + IntVar (pyout_v)
                - 5 CheckButton + IntVar (pyerr_v)
                - 5 CheckButton + IntVar (error_v)
                - 5 CheckButton + IntVar (no_output_v)
        - 3 Frame (result_main)
            - 4 Canvas (result_canvas)***note**
                - 5 Frame (frame_labels)***note**
                    - 6 Label (row_1 - rank)
                    - 6 Label (row_2 - score)
                    - 6 Label (row_12 - filename)
                    - 6 HTMLLabel (row_3 - github link)***note**
                    - 6 HTMLLabel (row_4 - local link)
                    - 6 HTMLLabel (row_5 - highlight text)
                    - 6 Label (row_6 - grey row)

                    (these seven widgets are repeated depending on the number of result of the query)
            - 4 Scrollbar (vsb)

Things to ***note** about the structure

- What is special about the *tk.frame "frame_labels"?*

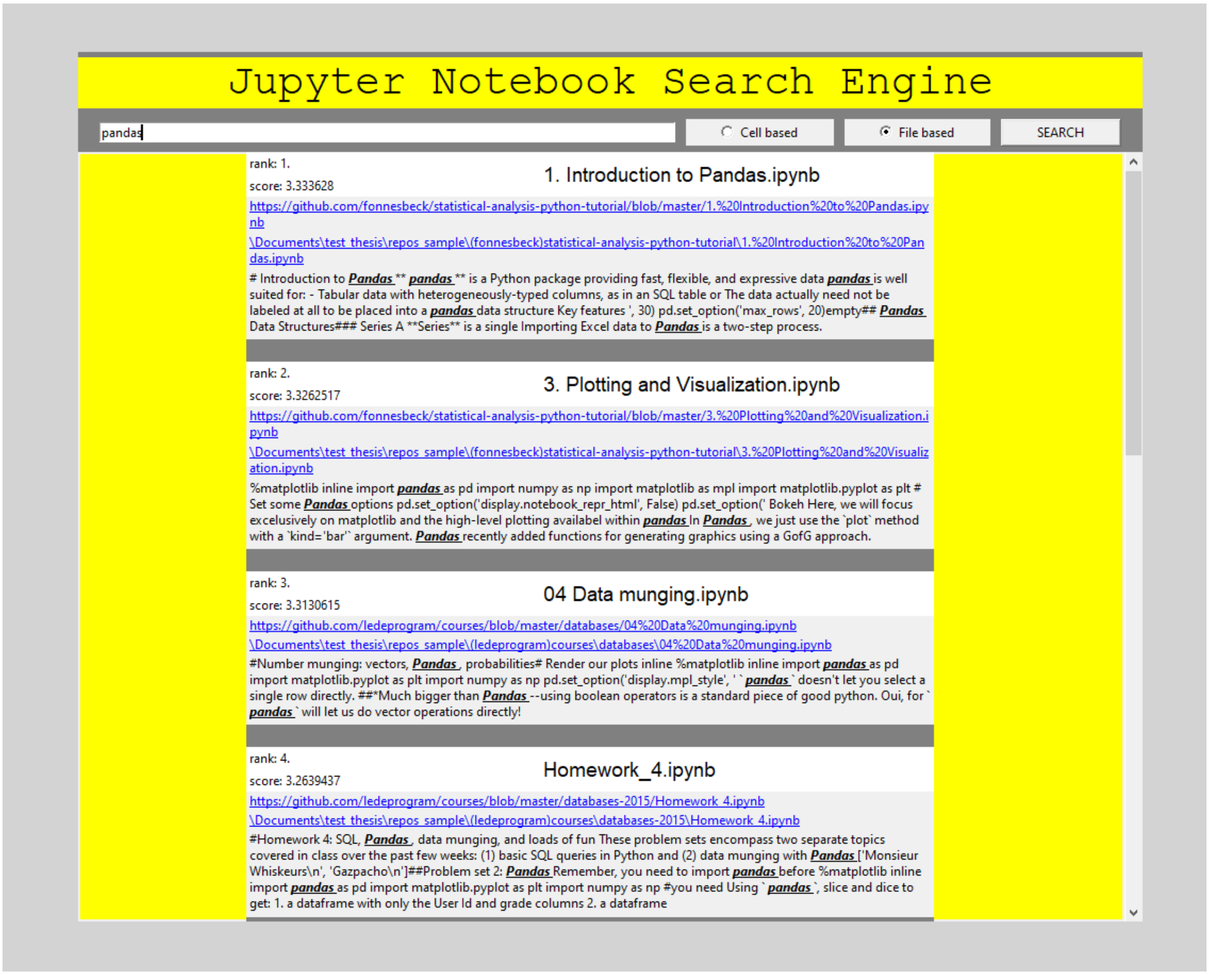
When creating this layout, all the widgets except the frame *frame_labels* are created one time only. The *frame_labels* is a variable that changes every time a new query is entered by the user by using the combination of the methods *slaves* and *forget*. The *grid_slaves* method is used to tell you all the widgets that are inside the *frame_labels* and the *grid_forget* method is used to remove those slaves.

- Why use a *tk.canvas* instead of *tk.frame?*

A tkinter's canvas primary use is to draw lines, graphs or plots. Since in this project there is no need for those actions, it may seem strange to see the use of a canvas. However, we will use a scrollbar and in order to scroll, the scrollbar needs to be associated with another widget in which the scrolling takes part. Unfortunately, the scrollbar cannot associate with the frame widget, however a commonly-used way is to use a canvas. Create the canvas and scrollbar within a frame (*f1*) and associate that scrollbar with the canvas. Then create a frame (*f2*) within that canvas and use the *create_window* method to set that frame (*f2*) as a window.

- Why use a *tk_html_widgets.htmllabel* instead of a *tk.label?*

In order to make the search engine more interactive and visually attractive in certain cases the widget standard Label was not able to offer enough. The module *tk_html_widgets* is a collection of tkinter widgets whose text can be set in HTML format (link), which includes the widget *HTMLLabel*. This widget makes it possible to style text, for example make text bold, and can create hyperlinks. The chosen structure will provide the following result below(see *figure 19*):



rank: 1.
score: 3.333628
1. Introduction to Pandas.ipynb
https://github.com/fonnesbeck/statistical-analysis-python-tutorial/blob/master/1.%20Introduction%20to%20Pandas.ipynb
\Documents\test_thesis\repos_sample\(fonnesbeck)statistical-analysis-python-tutorial\1.%20Introduction%20to%20Pandas.ipynb
Introduction to Pandas ** pandas ** is a Python package providing fast, flexible, and expressive data pandas is well suited for: - Tabular data with heterogeneously-typed columns, as in an SQL table or The data actually need not be labeled at all to be placed into a pandas data structure Key features ', 30) pd.set_option('max_rows', 20)empty## Pandas Data Structures### Series A **Series** is a single Importing Excel data to Pandas is a two-step process.

rank: 2.
score: 3.3262517
3. Plotting and Visualization.ipynb
https://github.com/fonnesbeck/statistical-analysis-python-tutorial/blob/master/3.%20Plotting%20and%20Visualization.ipynb
\Documents\test_thesis\repos_sample\(fonnesbeck)statistical-analysis-python-tutorial\3.%20Plotting%20and%20Visualization.ipynb
%matplotlib inline import pandas as pd import numpy as np import matplotlib as mpl import matplotlib.pyplot as plt # Set some Pandas options pd.set_option('display.notebook_repr_html', False) pd.set_option(' Bokeh Here, we will focus excelusively on matplotlib and the high-level plotting availabel within pandas In Pandas , we just use the `plot` method with a `kind='bar'` argument. Pandas recently added functions for generating graphics using a GofG approach.

rank: 3.
score: 3.3130615
04 Data munging.ipynb
https://github.com/ledeprogram/courses/blob/master/databases/04%20Data%20munging.ipynb
\Documents\test_thesis\repos_sample\(ledeprogram)courses\databases\04%20Data%20munging.ipynb
#Number munging: vectors, Pandas , probabilities# Render our plots inline %matplotlib inline import pandas as pd import matplotlib.pyplot as plt import numpy as np pd.set_option('display.mpl_style', ' ` pandas ` doesn't let you select a single row directly. ##*Much bigger than Pandas --using boolean operators is a standard piece of good python. Oui, for ` pandas ` will let us do vector operations directly!

rank: 4.
score: 3.2639437
Homework_4.ipynb
https://github.com/ledeprogram/courses/blob/master/databases-2015/Homework_4.ipynb
\Documents\test_thesis\repos_sample\(ledeprogram)courses\databases-2015\Homework_4.ipynb
#Homework 4: SQL, Pandas , data munging, and loads of fun These problem sets encompass two separate topics covered in class over the past few weeks: (1) basic SQL queries in Python and (2) data munging with Pandas ['Monsieur Whiskeurs\n', 'Gazpacho\n']##Problem set 2: Pandas Remember, you need to import pandas before %matplotlib inline import pandas as pd import matplotlib.pyplot as plt import numpy as np #you need Using ` pandas `, slice and dice to get: 1. a dataframe with only the User Id and grade columns 2. a dataframe

figure 19: SERP zoomed in on result_frame

4.5 Creating queries for elasticsearch

4.5.1 Elasticsearch DSL

Already known is that Jupyter Notebook is written in a json format, but the same applies for the queries that are used in Elasticsearch. Elasticsearch has created its own DSL (Domain Specific Language) based on JSON to define queries. The dsl can be split into nine categories of queries: compound, full text, geo, shape, joining, match all, span, specialized and term-level. The category that fits the needs of the search engine the best is the full text queries, since full text queries are used to search analysed text fields. Within the category *full text queries* the following nine types of queries exist: intervals, match, match_bool_prefix, match_phrase, match_phrase_prefix, multi_match, combined_fields, query_string and simple_query_string queries. For this project the following categories were interesting: match, multi_match, query_string.

Match is the standard query used to search within a text. It needs a text, number, boolean value or a date to search within a given field, after which it returns the documents that match with that. The given field in this project is one of the DataFrame columns that was indexed, for example the string of the cell. Match is used in the *JuNocta 0.9*.

Multi_match can be seen as an extension of *match*. It still needs a text, number, boolean value or a date to search, however it allows multiple fields to search within. Fields can individually be boosted to change the relevance of fields compared to each other, for example making a *title* field more relevant/important than a *description* field.

Query_string is the most complex out of the three categories. Query_string parses and splits the provided string based on/using its own syntax, which can be considered a 'mini-language'. *JuNocta 1.0* will make use of the following three attributes of this syntax: using parentheses and boolean operators to split a query, using double quotes to match with an exact phrase and specifying fields to search in. Examples of these three attributes are given below:

- (pandas OR pd) AND filter
- "memory statement"
- user:jakevdp AND spaced numbers

However, despite query_string being very versatile, the syntax of query_string is strict and can return an error if the provided string contains an invalid syntax. This can be seen as a downside, as a search engine that is 'as simple as possible' is desired for the user.

I initially used multi_match for *JuNocta 1.0*, as it provided the possibility to search multiple fields. However, I deemed the extra possibilities that query_string provides, more important than the 'as simple as possible' philosophy.

4.5.2 Query_string json structure

The structure of all queries in elasticsearch start with the same structure. The structure is a multidimensional json array that makes use of key-value pairs. In the first layer of this array, there is a key *query* and there might be other keys like *highlight*. The key *highlight* is optional and it indicates whether highlighted snippets of the matched text are returned. Since the layers that follow the key *highlight* seem self-explanatory, we will focus on the 'key' *query*. In the second layer the type of query is placed, examples of this would be *match*, *multi_match*, *query_string*, *bool* etc. What follows in the third layer is determined by the type of query that is chosen. In the case of *query_string* the third layer where the actual query is placed, called *query* (not to be confused with *query* in the first layer). The general structure of the query looks like *figure 20* below:

```
{
            'query': {
                        'query_string': {
                                             'query': ('variable x (user input)')
                                                       AND    (
                                                                 'variable y (cell vs file)'
                                                                 ),
                                            'default_field': 'string'
                                            }
             },
             'highlight':        {
                                  'pre_tags': ['<b>'],
                                  'post_tags': ['</b>'],
                                  'order': 'score',
                                  'fields': {
                                             'string': {}
                                             }
                                  }
}
```

Figure 20: structure of query

4.5.3 Difference between cell- and file-based

This query in figure *20* is divided into two variables: x and y. Variable x is the string that the user has typed in the Entry. Variable x is independent on whether the search is cell-based or file-based. Variable y are the variables that the user has checked off, this includes the radio buttons and check buttons within the menu buttons. Variable y is dependent on whether the search is cell-based or file-based. When cell-based, it will state the cell_types and output_types separately, by using the boolean operator 'AND' in between them. Within stating the different cell_types, it will use the boolean operator 'OR' (the same goes for the output_types). The reason for separating cell_types with output_types is to prevent a situation in which the *code* cell_type and an output_type are combined. For example, if not separated *'cell_type:code OR output_type:display_data'* could return any type of output_type, despite specifically selecting *display_data*. This is because all output types have *code* as their cell_type. By separating the and using the AND operator this is solved. The reason for using the OR-operator is when you look at a single cell, it can not have multiple cell_types (or output_types). An example of a cell-based query can be seen in figure 21, in which variable x is dark blue and variable y is dark green.

```
  'query': '
          (nested dataframe)
          AND (
               (cell_type:markdown)
               OR
               (cell_type:code)
             )
          AND (
               (output_type:execute_result)
               OR
               (output_type:display_data)
               )
          '
```

Figure 21: cell-based query

When file-based, it will combine the cell_types and output_types and uses AND-operator to do so. This causes that when a user selects certain multiple characteristics, the file must have all those characteristics included. An example of a file-based query can be seen in figure 22, in which variable x is dark blue and variable y is dark green.

```
'query': '
        (nested dataframe)
        AND (
            (code:1.0)
            AND (markdown:1.0)
            AND (stream:1.0)
            AND (execute_result:1.0)
            AND (display_data:1.0)
            )
        '
```

Figure 22: file-based query

4.6 Interaction Elasticsearch-Tkinter (GUI)

There are two ways to start the interaction between elasticsearch and tkinter: by clicking the search button or by hitting enter on the keyboard. Both these actions are binded to the function *search_clicked()*.

If either of these actions are performed, four variables will be collected: (1) the query typed in the entry, (2) whether the radiobutton is set on cell or file, (3) the cell types that are checked in the menubutton and (4) the output types that are checked in the other menubutton. If none of the checkbuttons in the menubutton are ticked, then they will be ignored.

After the variables are collected the function *query_string_cellbased()* or *query_string_filebased()* is called. Both functions create a query according to the elasticsearch dsl as described in the chapter *4.5 Creating queries for elasticsearch*. The created query is then passed to elasticsearch, elasticsearch returns results that are matching the query.

After the results are returned the function *display_row()* is called, this function loops through the results and displays them in the tk.Frame *frame_labels*. When a user enters a new query, the former results in *frame_labels* are removed and replaced by the new results.
[korte uitleg hoe het verwijzen naar een specifieke cel gaat door het url te veranderen]

4.7 Test methods

As established in the introduction the main research questions is: *In what ways does JuNocta improve searching for or within jupyter notebooks files compared to a search engine that only indexes the notebooks as text?* For each dataset the following sub questions are asked:

1. To what extent is it possible to run a search engine on your own laptop, that is designed for Jupyter notebooks and with just one push of a button can download, index entire sets of GitHub repositories and make the notebooks locally searchable?
    1. With what success rate can the repositories mentioned on the wiki page be cloned on/to the user’s computer (this question is only relevant to dataset 2)?
    2. With what success rate can the notebook files be indexed into a pandas DataFrame?
    3. With what success rate can the notebook files (pandas DataFrame) be inserted to Elasticsearch on the localhost/9200?
2. Is an advanced cell-type aware search engine preferable above a search engine that only indexes the notebooks as text?
    1. What is the precision of each search engine?
    2. What are the differences between using the two search engines in order to gather the information needs for each dataset, what are the possible advantages or disadvantages?

In order to compare the search engines, a user study has been created. The user study consist of 10 information needs for each dataset. The search engines will have their own query formulated. The search engine that only indexes the notebooks as text (JuNocta 0.9) will be represented as method 1 and the advanced cell-type aware search engine (JuNocta 1.0) will be represented as method 2. For method 2, it is indicated whether it searches on a cell- or file-based level with either *cell* > or *file* > at the start of each query. When cell types or code output types are select, those are displayed in parentheses with the word *select*. For example, (select markdown,code).

4.7.1 User study with 10 information needs

4.7.1.1 Dataset 1 - Jake Vanderplas books

Information need	**Query - method 1**	**Query - method 2**
1. How do you create a numpy array of spaced numbers?	numpy spaced numbers	cell > (numpy np*) AND spaced numbers
2. Can you find a file that explains the different magic and shell commands?	magic shell commands	file > (magic shell) AND command*
3. How do you index/range/filter a pandas DataFrame by time or date?	dataframe date range time	cell > (pandas pd*) AND (filter range index*) AND (time date)
4. How do you add a legend to a *matplotlib* plot?	matplotlib legend	cell > matplotlib AND legend AND (add OR create OR .)
5. How do you maximise the margin of a support vector machine with *sklearn*?	maximise margin support vector machine sklearn	cell > max* margin AND (svm* support vector machine) (sk* sklearn)
6. What is a generator expression?	generator expression	cell > generator expression
7. How do you use the *matplotlib* module in order to plot a three dimensional graph?	matplotlib three dimensional	cell > matplotlib AND three dimensional
8. How do you measure the memory use of a single statement?	memory single statement	cell > measure AND memory AND statement
9. Can you find a markdown cell that contains *lambda* in the Whirlwind Tour of Python book?	lambda	file > lambda AND repo:WhirlwindTourOfPython
10. Can you find a file that explains the different types of error in python?	error	file > error (select error)

4.7.1.2 Dataset 2 - Interesting gallery

Information need	Query - method 1	Query - method 2
1. How do you perform k-fold cross validation on a model?	k-fold cross validation	cell > k-fold AND cross validation
2. How do you create a heatmap with the module seaborn?	heatmap seaborn	cell > heatmap AND (seaborn sns*)
3. How do you turn a rgba scale into a grayscale for a plot?	rgba grayscale plot	cell > rgba AND grayscale
4. How do you get a summary of OLS regression results?	summary ols regression results	cell > summary AND ols regression results
5. How do you save a plot created with the module matplotlib?	matplotlib save imago figure	cell > matplotlib AND save AND (image OR figure)
6. How do you display tuples in a tabular form?	display tuples tabular format	cell > tuples AND tabular
7. Can you find a file that uses its kernels to run the programming language Haskell?	haskell	file > haskell (AND repo:IHaskell)
8. Can you find a cell that uses the *groupby()*, *transform()* and *lambda* in one line for a DataFrame?	dataframe groupby transform lambda	cell > *groupby AND *transform AND lambda
9. Can you find a file that analyses football data?	football soccer data	file > (football soccer) data
10. Can you find a file that provides information about the module *bqplot*?	bqplot	file > bqplot

693

694 4.7.1.3 Dataset 3 – EECN notebooks

Information need	Query - method 1	Query - method 2
1. How do you show the percentage within a *matplotlib* pie chart? (autopct)	percentage pie chart	cell > percentage AND pie chart
2. How do you flatten a nested json file to then be able to use in a pandas DataFrame?	flatten nested json pandas	cell > flatten AND nested json AND (pandas pd* dataframe df*)
3. How do you reshape a numpy array?	reshape numpy array	cell > reshape AND numpy array
4. How do you fill null values in a pandas DataFrame?	dataframe filling null values	cell > (pandas dataframe) AND filling null values
5. How do you decode base64 strings?	decode base64	cell > decode base64

6. How do you add an entry to a Menu in tkinter?	tkinter menu	cell > tkinter AND menu
7. What is a numpy eye?	numpy eye	cell > (numpy np*) AND eye
8. Can you find a file that covers the topic LSTM regarding machine learning?	lstm machine learning	file > "long short-term memory" (select code, markdown,heading)
9. Can you find a file that analyses data about the titanic?	titanic	file > titanic
10. Can you find a file that uses LSTM for text generation?	text generation lstm	file > text generation lstm

5. Evaluation

In this chapter the two sub questions of the main research question are evaluated. This is done by looking at the questions regarding each sub question. The first sub question is regarding the preparation of the dataset. The second sub question is regarding the testing of the developed search engine.

5.1 Research question 1

Research question 1 focusses on web scraping the wiki page, indexing data into a DataFrame and the insertion of data into Elasticsearch. The results of web scraping exceeded expectations. Not only did it get all the links referring directly to GitHub repositories, but I created extra links to repositories based on links referring to files or folders within a repository. This was achieved through a process of trial and error, while first I managed to extract around 100 repositories, I ended with the extraction of 211 repositories. Indexing the dataset into a DataFrame also turned out to go well, almost flawless. As the datasets grew bigger, the success rate got smaller marginally. The code itself proved to be flawless, but only a few files failed to index due to having faults in their JSON structure, which was out of my control to change. The insertion of data into Elasticsearch also went almost flawless. While the first dataset had a success rate of 100%, dataset 2 and 3 had a success rate of 99.9%. Although this seems already perfect, it can still be improved. If a bulk fails, a new *try and except* clause can be created in which the failed chunk will be split into smaller chunks that will be bulked separately. However, I deemed this not as important. Although it now seems that everything went perfectly, it should be noted that all the coding regarding dataset 3 went not smoothly. The process of indexing and inserting took around 100 minutes. If adjustments had to be made afterwards, this process needed to run again, which would take a lot of time.

5.1.1 With what success rate can the repositories mentioned on the wiki page be cloned on/to the user's computer?

This question is only relevant to dataset 2. There are five stages in which the links to GitHub repositories are web scraped, trimmed down to actual GitHub repositories urls and finally gets cloned.

Stage 1

On the wiki page there are a total of 852 links in the body, however there are multiple duplicates that need to be removed. Of the 852 links 754 remain as unique, meaning 88.5% of all the links in the body are unique.

Stage 2

The four categories of links mentioned in chapter *4.2.1 Web Scraping* are:

1. a url that contained the string *github* (415 links).
2. a hash-link to a specific part within the wiki page (43 links).
3. a url of a nbviewer of which there was no way to extract a github repository (62 links).
4. a url of a different site (234 links).

As described, of those four categories, only links that fall under the first category are continued with. So of the 754 unique links 415 remain as github links, meaning 55.0% of all the links in the body were github related.

Stage 3

The three github url patterns mentioned in chapter *4.2.1.1 Github url pattern* are:

1. url of a repository (108+4=112 links).
2. url of a certain file or directory within a repository (30+168=198 links).
3. url of a github user (66 links).
4. patterns that did not match with the above, but contained the string ‘github’ (39 links)

As described, of those four patterns, only urls according to pattern 1 and 2 are continued with. Urls of pattern 2 are transformed to urls of repositories using regular expressions. The urls of patterns 1 and 2 are combined 310, but when removing duplicates 224 remain. Duplicates are possible after transformations of urls (pattern 2), meaning multiple urls to files within the same repository become duplicate urls of the same repository. Of the 415 links 224 remain as GitHub repositories, meaning 54.0% of all the github related links were unique and usable.

Stage 4

As mentioned in the same chapter, there do exist links to repositories that do not exist anymore and show a webpage with “*Page not found*” as its only content. There were 8 links to non-existing repositories. There are two reasons for this: either the link actually used to be a repository or the transformation of a url created a false repository that never existed. The first reason caused 6 faulty links and the second reason caused 2 faulty links. Of the 224 links 216 remain as existing GitHub repositories, meaning 96.4% are actual existing repositories.

Stage 5

Finally, when cloning these 216 repositories 211 are cloned successfully, meaning the cloning itself had a success rate of 97.7%.

Now if we revisit the question: *With what success rate can the repositories mentioned on the wiki page be cloned on/to the user’s computer?* As established, there were 224 unique GitHub mentioned on the wiki page of which 211 are cloned, meaning a success rate of 94.2%.

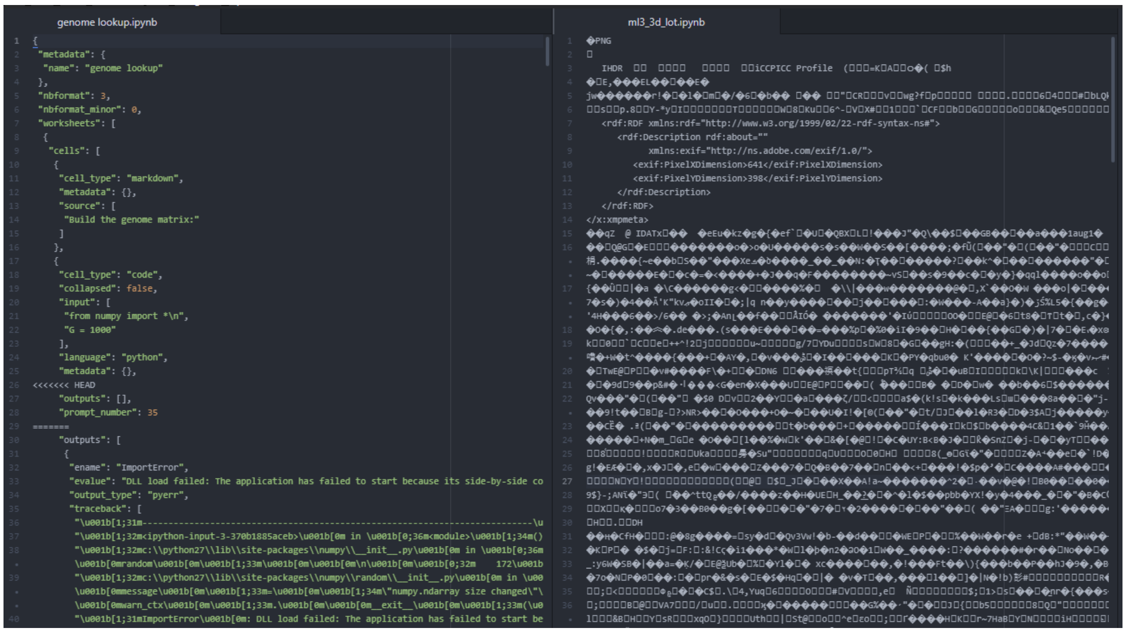
5.1.2 With what success rate can the notebook files be indexed into a pandas DataFrame?

Dataset 1

Dataset 1 contains 86 notebooks of which 86 are indexed without any problems, meaning a success rate of 100%.

Dataset 2

As established in *research question 5.1.1* dataset 2 consist of 211 repositories successfully cloned to the user’s computer. These repositories contain 3132 notebooks of which 3130 are indexed without any problems, meaning a success rate of 99.9%. The two notebooks that caused problems had a corrupt json structure (see figure 23). The left notebook had a json.decoder.JSONDecodeError, with the reason *Expecting property name enclosed in double*, which can be seen at line 26. The right notebook had an UnicodeDecodeError, with the reason *utf-8, can't decode invalid start byte*.



```
1  {
2   "metadata": {
3    "name": "genome lookup"
4   },
5   "nbformat": 3,
6   "nbformat_minor": 0,
7   "worksheets": [
8    {
9     "cells": [
10     {
11      "cell_type": "markdown",
12      "metadata": {},
13      "source": [
14       "Build the genome matrix:"
15      ]
16     },
17     {
18      "cell_type": "code",
19      "collapsed": false,
20      "input": [
21       "from numpy import *\n",
22       "G = 1000"
23      ],
24      "language": "python",
25      "metadata": {},
26 <<<<<<< HEAD
27      "outputs": [],
28      "prompt_number": 35
29 =======
30      "outputs": [
31       {
32        "ename": "ImportError",
33        "evalue": "DLL load failed: The application has failed to start because its side-by-side co
34        "output_type": "pyerr",
35        "traceback": [
36         "\u001b[1;31m---------------------------------------------------------------------------\u
37         "\u001b[1;32m<ipython-input-3-370b1885aceb>\u001b[0m in \u001b[0;36m<module>\u001b[1;34m()
38         "\u001b[1;32mc:\\python27\\lib\\site-packages\\numpy\\__init__.py\u001b[0m in \u001b[0;36m
 .        \u001b[0mrandom\u001b[0m\u001b[1;33m\u001b[0m\u001b[0m\n\u001b[0m\u001b[0;32m    172\u001b
39         "\u001b[1;32mc:\\python27\\lib\\site-packages\\numpy\\random\\__init__.py\u001b[0m in \u00
 .        \u001b[0mmessage\u001b[0m\u001b[1;33m=\u001b[0m\u001b[1;34m\"numpy.ndarray size changed\"\
 .        \u001b[0mwarn_ctx\u001b[0m\u001b[1;33m.\u001b[0m\u001b[0m__exit__\u001b[0m\u001b[1;33m(\u0
40         "\u001b[1;31mImportError\u001b[0m: DLL load failed: The application has failed to start be
```

```
1  �PNG
2  
3     IHDR              iCCPICC Profile  (   =K A  o�(  $h
4  � E,���EL�� ���E�
5  jw[illegible]
6  [illegible]
7     <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
8        <rdf:Description rdf:about=""
9              xmlns:exif="http://ns.adobe.com/exif/1.0/">
10          <exif:PixelXDimension>641</exif:PixelXDimension>
11          <exif:PixelYDimension>398</exif:PixelYDimension>
12       </rdf:Description>
13    </rdf:RDF>
14 </x:xmpmeta>
15 ��qZ  @ IDATx[illegible]
[illegible]
```

figure 23: two notebooks with corrupt json structure making it not indexable (notebook on left: corrupt at line 26 vs notebook on right: whole file corrupt)

Dataset 3

Dataset 3 contains 198.778 notebooks of which 196.267 are indexed without any problems, meaning **a** success rate of 98.7%. 2554 notebooks failed to index, not surprisingly as there are already 454 notebooks with a size of 0kb. There are multiple reasons for notebooks to fail, the method *Counter* (from collections) gave an insight to these reasons. These are the four types of errors, when indexing the notebooks, with the number of occurrences added:

1. 2430 json.decoder.JSONDecodeError
2. 61 KeyError
3. 18 TypeError
4. 2 IndexError

When an error occurs, a tuple containing the reason for the error and its location (combining the line, column and character) is relayed. This is the top 3 most common error tuples, with the number of occurrences added:

1.	1169	Expecting value: line 1 column 1 (char 0)	(meaning a blank file)
2.	788	Extra data: line 1 column 4 (char 3)	(file with only the string *404: Not Found*)
3.	90	Expecting value: line 2 column 1 (char 1)	(blank file, but with a newline)

This is the top 3 most common reasons for an error, with the number of occurrences added:

1. 1327 Expecting value
2. 798 Extra data
3. 210 Expecting property name enclosed in double quotes ()

Of the 2511 errors there are 268 unique tuples, containing 18 unique reasons (see appendix). For example, 1327 cases for the reason *Expecting value* of which 1169 are at line:1/column:1/char:0 meaning it is a blank file.

5.1.3 With what success rate can the notebook files (pandas DataFrame) be inserted to elasticsearch on the localhost/9200?

Since *JuNocta* works on a cell and file-based level, two DataFrames need to be created for each dataset and both need to be inserted into Elasticsearch on the localhost/9200.

Dataset 1

On a cell-based level, the DataFrame contains 3781 rows of which 3781 are inserted without any problems, meaning a success rate of 100%.
On a file-based level, the DataFrame contains 86 rows of which 86 are inserted without any problems, meaning a success rate of 100%. Combining both percentages would mean **a success rate of 100%**.

Dataset 2

On a cell-based level, the DataFrame contains 108.110 rows of which 108.085 are inserted without any problems, meaning a success rate of 99.98%. 25 rows were unsuccessful. The reasons for these failures are unknown. The inserting of the DataFrame into elasticsearch was done by looping through chunks of the dataframe and using a *try and except* clause within that loop. If a chuck would fail, it would store the exception and location of the chuck to a dictionary. However, no exceptions were saved to the dictionary.
On a file-based level, the DataFrame contains 3130 rows of which 3130 are inserted without any problems, meaning a success rate of 100%.
Combining both percentages would mean a success rate of either their average of the percentages, which is 99.99% or percentage of the rows combined, which is 99.98%.

Dataset 3

On a cell-based level, the DataFrame contains 5.324.962 rows of which 5.324.128 are inserted without any problems, meaning a success rate of 99.98%. 834 rows were unsuccessful. The reasons for this failure were either a *ReadTimeoutError(HTTPConnectionPool(host='localhost', port=9200)*, which occurred 49 times and means the request takes longer process than the default time, or a *TransportError(413, '', None)*, which occurred once and means the request entity is too large.
On a file-based level, the DataFrame contains 196.267 rows of which 195.767 are inserted without any problems, meaning a success rate of 99.75%. 500 rows were unsuccessful, with *TransportError(413, '', None)* as its reason.
Combining both percentages would mean a success rate of either their average of the percentages, which is 99.86% or percentage of the rows combined, which is 99.98%.

5.2 Research question 2

Research question 1 focusses the testing of the search engine, this was conducted through a user case, in which 10 information needs were formulated. For each information need, the top 10 results for both methods needed to be classified as relevant or not relevant, this meant that a total of 516 results needed to be classified. This proved to be more time consuming than I thought. The testing revealed that the average precision of method 2 (*JuNocta 1.0*) was higher than method 1 (*JuNocta 0.9*) for each dataset. Coincidentally the precision of method 2 was 0.80 for each dataset. There are only two instances in which the precision of a single query is higher for method 1 than method 2. The overall higher precision of method 2 is due to its options to make the query specific. Although beforehand, I thought the option to select different cell types or code output types would have the highest impact on the precision, testing proved otherwise. Making use of the query_string syntax proved to be most beneficial. This does not negate the relatively large amount of time spent making those functions, as they still can be very useful. Classifying results for dataset 1 and 2 was straightforward and produced accurate precisions, however I deem the precisions by dataset 3 not completely accurate. This is due to the fact that dataset 3 has multiple identical or almost identical files. For example, there were two extreme situations in which a top 10 result could consist of 9 duplicates. If these were relevant a precision of at least 0.9 was achievable, otherwise a precision with a maximum of 0.1 was achievable. Either way, this would have a negative effect on the precision. At last, the natural language in which the markdown cells were written had effect on the precision. I deemed cells in written in Spanish, Chinese or Russian not relevant as I was not able to understand and therefore could not satisfy the information need.

5.2.1 What is the precision of each method?

For each dataset a table is shown with the relevance of the top 10 results for each query for both methods. There are instances in which there are not ten results, but for example only four results. When searching for a cell, if the cell itself or surrounding cells provide an answer to the information need the result is considered relevant. For example, the cell is a markdown cell with explanation and below the cell is a code cell with the answer to the information need. When searching for a file, if the file itself or somewhere in the file an answer is provided the result is considered relevant. In the table, if a result was relevant it is displayed as R, if a result was not relevant it is displayed as N. Besides the relevance, the precision at 10 (p) has been calculated and is displayed as a column for each method in the table. Precision at 10 is the ratio of relevant hits among the first maximally 10 returned search results. If in a situation when there are less than 10 search results are returned, the ratio of relevant hits among returned search results is calculated. One could argue that in this situation, the amount of relevant returned search results should be divided and therefor the search engine would be punished for retrieving less result than 10. However, I encourage the search engine to return less results when the query is really specific, especially in the case of a small dataset. That is why I will not divide by 10 regardless, but maximally by 10. If the precision for a method is higher than the other, it is shown in bold. Below each table the averaged precision for each method has been calculated, after which an independent samples t-test has been calculated to verify whether the difference was significant. A significance value than 0.05 is needed to indicate a difference.

Dataset 1 - Jake Vanderplas books

Information need	Query - method 1	p	Query - method 2	p
1. How do you create a numpy array of spaced numbers?	R,N,N,N,N,N,N,N,N,N	0.1	R,N,N	**0.3**
2. Can you find a file that explains the different magic commands?	R,N,R,R,R,N,N,R,N,N	0.5	R,R,R,N,R,R,R,R,R,N	**0.8**
3. How do you filter a pandas DataFrame by date?	N,N,N,N,R,R,N,N,N,N	0.2	R,R,N,N,N	**0.4**
4. How do you add a legend to a *matplotlib* plot?	R,R,R,R,R,R,R,R,R,R	1.0	R,R,R,R	1.0
5. How do you maximise the margin of a support vector machine with *sklearn*?	R,R,N,N,N,R,N,R,R,N	0.5	R,R,R,R,R,R	**1.0**
6. What is a generator expression?	R,R,R,R,R,R,R,R,R,R	1.0	R,R,R,R,R,R,R,R,R,R	1.0
7. How do you use the *matplotlib* module in order to plot a three dimensional graph?	R,R,R,R,R,R,R,R,R,R	1.0	R,R,R,R,R,R,R,R,R,R	1.0
8. How do you measure the memory use of a single statement?	R,N,N,N,N,N,R,N,N,N	0.2	R	**1.0**

9. Can you find a markdown cell that contains *lambda* in the Whirlwind Tour of Python book	N,R,N,N,N,N,N,R,N,R	0.3	R,R,R,R,R,R	**1.0**
10. Can you find a file that explains the different types of error in python?	R,N,R,N,N,N,R,N,R,N	0.4	R,R,R,N,N,N,N	**0.4**

The queries for method 1 had an averaged precision of 0.52. The queries for method 2 had an averaged precision of 0.7962. Method 2 proved better in seven situations and was equal in three situations compared to method 1. This proves not to be a significant difference as the significance value is 0.073.

Dataset 2 - Interesting gallery

Information need	Query - method 1	p	Query - method 2	p
1. How do you perform k-fold cross validation on a model?	R,R,R,R,R,R,R,R,R,R	1.0	R,R,R,R,R,R,R,R,R,R	1.0
2. How do you create a heatmap with the module seaborn?	N,N,N,N,N,N,N,N,N,N	0.0	R	**1.0**
3. How do you turn a rgba scale into a grayscale for a plot?	R,N,N,N,R,N,R,R,R,N	**0.5**	R,N,N	0.3
4. How do you get a summary of OLS regression results?	R,N,R,R,N,R,R,N,R,R	0.7	R,R,R,R,R,R,R,R,R,R	**1.0**
5. How do you save a plot created with the module matplotlib?	R,N,N,R,R,N,R,N,N,N	0.4	R,R,R,R,N,N,N	**0.6**
6. How do you display tuples in a tabular form?	R,N,N,N,N,N,N,N,N,N	0.1	R	**1.0**
7. Can you find a file that uses its kernels to run the programming language Haskell?	R,N,R,N,N,R,N,R,N,N	0.4	R,R,R,N,N,R,N,R,R,R	**0.7**
8. Can you find a cell that uses the *groupby()*, *transform()* and *lambda* in one line for a DataFrame?	N,N,N,N,N,N,N,N,N,N	0.0	R,N,N,R	**0.5**
9. Can you find a file that analyses football data?	N,R,R,R,R,N,N,N,R,R	0.6	R,R,R,R,N,N,R,R,R,R	**0.8**
10. Can you find a file that provides information about the module *bqplot*?	R,N,R,R,R,R,N,N,N,N	0.5	R,R,R,R,R,R,R,R,R,R	**1.0**

The queries for method 1 had an averaged precision of 0.42. The queries for method 2 had an averaged precision of 0.80. Method 2 proved better in eight situations, was equal in one situation and worse in one situation compared to method 1. This proves to be a significant difference as the significance value is 0.008.

Dataset 3 – EECN notebooks

Information need	Query - method 1	p	Query - method 2	p
1. How do you show the percentage within a *matplotlib* pie chart?	R,N,N,N,N,N,N,N,N,N	0.1	R,R,R,R,R,R,R,R,R,N	**0.9**
2. How do you flatten a nested json file to then be able to use in a pandas DataFrame?	N,R,R,R,R,R,R,R,R,R	**0.9**	R,N,R,N,N	0.4
3. How do you reshape a numpy array?	N,R,R,R,N,R,R,R,R,N	0.7	N,R,N,N,R,R,R,R,R,R	0.7
4. How do you fill null values in a pandas DataFrame?	R,R,R,N,N,N,N,N,N,N	0.3	R,R,R,N,R,R,R,R,R,R	**1.0**
5. How do you decode base64 strings?	R,R,R,R,R,R,R,R,R,R	1.0	R,R,R,R,R,R,R,R,R,R	1.0
6. How do you add an entry to a Menu in tkinter?	R,R,R,N,N,R,N,N,N,N	0.4	R,R,R,R,R,R,N,R,R,R	**0.9**
7. What is a numpy eye?	N,N,N,R,R,R,N,N,R,N	0.4	R,R,R,N,N,N,N,N,R,R	**0.5**
8. Can you find a file that covers the topic LSTM regarding machine learning?	R,N,N,R,R,N,N,N,N,N	0.3	R,R,R,N,R,R,N,N,N,R	**0.6**
9. Can you find a file that analyses data about the titanic?	R,N,N,R,R,R,R,R,R,R	0.8	R,R,R,R,R,R,R,R,R,R	**1.0**
10. Can you find a file that uses LSTM for text generation?	R,R,R,R,N,N,N,R,N,N	0.5	R,R,R,R,R,R,R,R,R,R	**1.0**

The queries for method 1 had an averaged precision of 0.54. The queries for method 2 had an averaged precision of 0.80. Method 2 proved better in seven situations, was equal in two situations and worse in one situation compared to method 1. This proves to be a significant difference as the significance value is 0.042. When combining the precisions of all the datasets for method 1 and combining the precisions of all the datasets for method 2, there is a significant difference between the two methods. The significance value is 0.0001.

5.2.2 What are the differences between using the two methods in order to gather the information needs for each dataset, what are the possible advantages or disadvantages?

Method 2 makes use of the syntax that is included with *query_string* as described in *4.5 Creating queries for Elasticsearch*. This makes it possible to create queries for really specific information needs by using one of the following three attributes: using parentheses and Boolean operators to split a query, using double quotes to match with an exact phrase and specifying fields to search in. These attributes can be used individually or combined in a query. Therefore, if a preferred answer is not found method 2 still provides a lot of room for the user to tweak the query and search again to hopefully get the preferred answer. This is opposed to method 1 has no influence, as it can only change the order of words.

As shown in relevance tables, method 1 will give 10 results no matter what. In contrast to method 2 that while being specific will only give results that match those criteria. With method 2, as an information need tends to get more specific, the results get fewer. For example, information need 8 of dataset 1, method 1 returns 10 results of which the first and the seventh are deemed relevant, while method 2 returns only one result which is deemed relevant. This gives the results provided by method 2 more value, as they tend to have a higher chance of being relevant (precision).

6. Conclusion & discussion

If we revisit the main research question, where is asked for the ways in which *JuNocta* does improve the searching in or for notebook files compared to a search engine that only indexes the notebooks as text. At the moment of writing, there is not a way to search within a collection of notebook files stored locally. File explorers of different operating systems, for example windows, only allow searching for filenames. For the matter of online notebooks, GitHub does provide an advanced search for all its file types. Although it does provide the option to filter notebooks files by making use of the file extension filter (.ipynb), these advanced filters mainly relate to GitHub characteristics. For example, the user or how many stars a repository got. Both searching using a file explorer or using the advanced search of GitHub, are in no comparison with the developed *JuNocta* and its described features. When comparing both search engines, the advanced cell-type aware search engine proved to have significantly higher precision. However, the higher precision might come at the cost of having a good understanding of the query_string syntax. It takes more time to formulate each of those complex queries.

7. Future works

Improvements to the search engine could be made to the indexing of data and the SERP in which interaction with the user takes place.

How bigger the collection of notebooks becomes, the higher the chance of duplicate code. Karen and Wrigstad (2020) looked at the characteristics of jupyter notebook code found on GitHub, they drew two conclusions from this. Of all code snippets 70% is duplicate and that around 50% of notebooks do not have any unique snippets. This has been confirmed in the testing of dataset 3, which consists of around 200.000 files scraped from GitHub and has shown a lot of duplicate files/cells in the results of the queries. Therefore, it would be a good idea to look at identical files and identical cells within the corpus and combine them, for example that would mean that a file would have multiple users or that a cell could have multiple file names. It would be interesting to see if this altercation of the dataset will have an effect on the precision.

In this search engine the focus is on either cell- or file-level, but what if the focus is on line-level. On this level, regular expressions could detect comments or imports. Different weights could be given each type of line, for example a “import numpy as np” could be given a low weight. When a line is detected in which a module is imported, it could receive extra attention. On a file-level, a file could get a label of the module that is used. On a cell- and line-level, the cells or lines below import-cell could be checked for use of that module. For example, with the line ‘import matplotlib.pyplot as plt’ other cells could be checked on a line-level for use of ‘plt.*’. However, this does not detect an import that makes use of the ability to import all names that a module defines. For example, “from bqplot import *”. On a cell-level the search engine could also be improved by indexing the surrounding cells, for example take cell *c*. The search engine should look at two cells above and two cells below cell *c* and give those surrounding cells a lower weight than cell *c*. This would mean if a query term resides in a cell above cell *c*, *c* would get a higher score. However, would significantly increase the size of the dataset and could cause problems as it could increase the request time or the size of the request entities.

When testing and searching for code specific needs, it struck me that a lot of code lines were split up in undesirable parts, to the extent in which a user needs to adjust the query to find a desirable answer. For example the line *df.groupby('key').transform(lambda x: x - x.mean())*, in a ideal situation this would split in six parts: *df / .groupby() / 'key' / .transform() / lambda / x:x-x.mean()*. However at the moment, the current tokenization will split it up into ten parts: *df.groupby / ('key'). / transform / (/ lambda / x: / x / - / x.mean())*. It would be interesting to see in what ways the tokenization can be adjusted, to achieve more desirable splits of code lines.

However, the biggest improvement would be letting JuNocta update the corpus itself automatically if a file has been edited. This improvement would make it 'ready to use' for a lot of students who have many notebooks lost somewhere in their directory. This in combination with the option to choose the max_size of the results as a variable in the SERP or the option to go to a next page with ten results.

When displaying the highlight of the result, the actual line and immediately surrounding lines could be shown. This is different from the current situation where highlights from different lines within the cell (or file) are pasted together. Finally, a minor update would be a slider for the ratio (or a set of a few predetermined ratios) of markdown-code cells within a file when the user is searching on a file-based level. During testing when the information need was an inquiry about learning regarding a specific topic. When the user is new to a topic, he might want relatively more explanatory markdown cells. When the user is very familiar with the topic, he might want more code cells.

8. Bibliography

- Chatterjee, S., Juvekar, S., & Sen, K. (2009, March). Sniff: A search engine for java using free-form queries. In International Conference on Fundamental Approaches to Software Engineering (pp. 385-400). Springer, Berlin, Heidelberg.
- Head, A., Hohman, F., Barik, T., Drucker, S. M., & DeLine, R. (2019, May). Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (pp. 1-12).
- Friedman, N. (2021, November 3). Thank you, GitHub. The GitHub Blog. Retrieved May 4, 2022, from https://github.blog/2021-11-03-thank-you-github/
- Imminni, S. K., Hasan, M. A., Duckett, M., Sachdeva, P., Karmakar, S., Kumar, P., & Haiduc, S. (2016, May). Spyse-a semantic search engine for python packages and modules. In 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C) (pp. 625-628). IEEE.
- Källén, M., & Wrigstad, T. (2020). Jupyter Notebooks on GitHub: Characteristics and Code Clones. arXiv preprint arXiv:2007.10146.
- Kim, K., Kim, D., Bissyandé, T. F., Choi, E., Li, L., Klein, J., & Traon, Y. L. (2018, May). FaCoY: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering* (pp. 946-957).
- Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B. E., Bussonnier, M., Frederic, J., ... & Willing, C. (2016). Jupyter Notebooks-a publishing format for reproducible computational workflows (Vol. 2016, pp. 87-90).
- Perkel, J. M. (2018). Why Jupyter is data scientists' computational notebook of choice. Nature, 563(7732), 145-147.
- Pimentel, J. F., Murta, L., Braganholo, V., & Freire, J. (2019, May). A large-scale study about quality and reproducibility of jupyter notebooks. In 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR) (pp. 507-517). IEEE.
- Rule, A., Tabard, A., & Hollan, J. D. (2018). Data from: Exploration and explanation in computational notebooks. UC San Diego Library Digital Collections.
- Rule, A., Birmingham, A., Zuniga, C., Altintas, I., Huang, S. C., Knight, R., ... & Rose, P. W. (2019). Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks. PLoS computational biology, 15(7), e1007007.

9. Appendix

All code can be found on the GitHub repository here[https://github.com/Kennitos/search_engine]