Project 2: Multi-source code search

Ardigò Susanna

December 7, 2020

https://github.com/SusyPinkBash/multi_source_code_search

Contents

1 Data Extraction

1.1 Goal and Input parameter

This part of the project consists of extracting names and comments of Python classes, methods and functions and save them in a csv file.
This file takes as argument the path of the directory of the project that we want to analyze. For this project we use the project `tensorflow`.

1.2 Description of the code

To efficiently parse the files in the directory, we created a class named `Visitor`, which extends the `NodeVisitor` class of the standard library ast (which stands for Abstract Syntax Tree). This class holds the path of the file. There is a global variable `data` used throughout the execution to store all the information extracted.
The function `start(directory_path)` *'walks'* the given directory using the function `walk` which generates a 3-tuple of directory path, directory names and file names. We open and read all the python files, checked with the extension of the file, we create a Visitor object and start to visit. The class we created has two different visit methods which differ in if the node visiting is a definition of a class or a function.
The method `visit_FunctionDef(self, node: FunctionDef)` adds the node information to the array of data if the function or method is not a main or a test. Since this method is used both for functions and methods, we know that is a method if the first argument is self. The method `visit_ClassDef(self, node: ClassDef)` calls a generic visit (of the ast library) and, as the previous method, adds the node information to the array of data if the class is not a main or a test.
After the parsing is complete I create a pandas dataframe, feeding it as data the data array, and export it in a csv extension.

1.3 Results

Table 1 show the number of Python files, classes, methods and functions found while parsing the Tensorlow directory.

Type	#
Python files	2817
Classes	1904
Methods	7271
Functions	4881

Table 1: Count of data found in Tensorflow

2 Training of search engines

2.1 Goal and Input parameter

This part of the project consists of representing code entities using the four embeddings frequency, TF-IDF, LSI and Doc2Vec.
This file takes as argument a query.

2.2 Description of the code

The function `start(query)` loads the csv into a pandas dataframe and then computes the results. The first part of function `compute_results(query, dataframe)` creates the necessary data and normalize the query that the second part needs to produce the results. The first part of function `create_data(dataframe)` extracts the names and comments from the dataframes to create a clean array of arrays of tokens and a dictionary with the frequencies of each token. In the second part we create the corpus by processing the tokens, we create a gensim dictionary and the bag of words. In the second part of function `compute_results(query, dataframe)` we create a dictionary that hold the results of the searches and a dictionary to save the embedding vectors.
The function `query_frequency(query, bow, dictionary)` creates a sparse matrix of the bag of words and returns an array with the similarity scores of each entity of the given csv file. This array is then filtered to extract only the top 5 scoring entities. Similarly, the function `query_tfidf(query, bow, dictionary)` creates a sparse matrix of the tfidf model of the bag of words and returns an array with the similarity scores which is then filtered. The function `query_lsi(query, bow, dictionary)` creates a lsi model based on the bag of words, a vector based on the model and the dictionary, the matrix of the similarities and the embedding vectors. The result of the matrix, as in the previous cases, is filtered to get only the top 5 scores. The function `query_doc2vec(query, bow, dictionary)` creates a doc2vec model which then feed the corpus to and train it. We save the trained model in an external pickle file to load it in the next runs. This improves the running time of this function. We create a vector infering it from the query, we create the similarity and take only the top 5 scores and the embedding vectors.
We create a dataframe with the information stored in the dictionary, we print the results and save them in a separate file.

2.3 Results

To show the results we run this part of the project with the query: *'Optimizer that implements the Adadelta algorithm'*.
Figure 1 show the result of the given query. As we can see in the image all results are classes. Almost all results have as comment the sentence *'Optimizer that implements the x algorithm'* with x being an algorithm name.

The most common document found is `Adadelta` with 3 findings, `AdadeltaOptimizer` with 3. Both documents have the same comments. The first, with path ../tensorflow/tensorflow/python/keras/optimizer_v2/adadelta.py, was found by Frequencies, TD-IDF and LSI. The second, with path ../tensorflow/tensorflow/python/training/adadelta.py, was found by Frequency, TF-IDF and Doc2Vec. Frequency and TF-IDF have the best results.

	name	file	line	type	comment	search
1	Nadam	../tensorflow/tensorflow/python/keras/optimizer_v2/nadam.py	34	class	Optimizer that implements the NAdam algorithm.	FREQ
2	Adadelta	../tensorflow/tensorflow/python/keras/optimizer_v2/adadelta.py	32	class	Optimizer that implements the Adadelta algorithm.	FREQ
3	FtrlOptimizer	../tensorflow/tensorflow/python/training/ftrl.py	29	class	Optimizer that implements the FTRL algorithm.	FREQ
4	AdagradOptimizer	../tensorflow/tensorflow/python/training/adagrad.py	32	class	Optimizer that implements the Adagrad algorithm.	FREQ
5	AdadeltaOptimizer	../tensorflow/tensorflow/python/training/adadelta.py	29	class	Optimizer that implements the Adadelta algorithm.	FREQ
6	Adadelta	../tensorflow/tensorflow/python/keras/optimizers.py	383	class	Adadelta optimizer.	TF-IDF
7	Nadam	../tensorflow/tensorflow/python/keras/optimizer_v2/nadam.py	34	class	Optimizer that implements the NAdam algorithm.	TF-IDF
8	Adadelta	../tensorflow/tensorflow/python/keras/optimizer_v2/adadelta.py	32	class	Optimizer that implements the Adadelta algorithm.	TF-IDF
9	AdamOptimizer	../tensorflow/tensorflow/python/training/adam.py	32	class	Optimizer that implements the Adam algorithm.	TF-IDF
10	AdadeltaOptimizer	../tensorflow/tensorflow/python/training/adadelta.py	29	class	Optimizer that implements the Adadelta algorithm.	TF-IDF
11	RMSprop	../tensorflow/tensorflow/python/keras/optimizer_v2/rmsprop.py	35	class	Optimizer that implements the RMSprop algorithm.	LSI
12	Adamax	../tensorflow/tensorflow/python/keras/optimizer_v2/adamax.py	33	class	Optimizer that implements the Adamax algorithm.	LSI
13	Ftrl	../tensorflow/tensorflow/python/keras/optimizer_v2/ftrl.py	30	class	Optimizer that implements the FTRL algorithm.	LSI
14	Adam	../tensorflow/tensorflow/python/keras/optimizer_v2/adam.py	34	class	Optimizer that implements the Adam algorithm.	LSI
15	Adadelta	../tensorflow/tensorflow/python/keras/optimizer_v2/adadelta.py	32	class	Optimizer that implements the Adadelta algorithm.	LSI
16	Adam	../tensorflow/tensorflow/python/keras/optimizer_v2/adam.py	34	class	Optimizer that implements the Adam algorithm.	Doc2Vec
17	AdagradOptimizer	../tensorflow/tensorflow/python/training/adagrad.py	32	class	Optimizer that implements the Adagrad algorithm.	Doc2Vec
18	AdamOptimizer	../tensorflow/tensorflow/python/training/adam.py	32	class	Optimizer that implements the Adam algorithm.	Doc2Vec
19	MomentumOptimizer	../tensorflow/tensorflow/python/training/momentum.py	29	class	Optimizer that implements the Momentum algorithm.	Doc2Vec
20	AdadeltaOptimizer	../tensorflow/tensorflow/python/training/adadelta.py	29	class	Optimizer that implements the Adadelta algorithm.	Doc2Vec

Figure 1: Results of the given query

3 Evaluation of search engines

3.1 Goal and Input parameter

This part of the project consists of measuring the precision and recall given 10 queries along with their ground truth.
This file takes as argument the path of the ground truth file.

3.2 Description of the code

The function `start(path_ground_truth)` loads the csv of the data into a pandas dataframe, parses the ground truth and then computes the precision and recall.
To efficiently parse the ground truth file, we created a class named `Truth` which holds the name, path and query. We read the ground truth file and create an array with all the entries of the ground truth and the queries.
To compute precision and recall we get the data of the results and the embedding vectors from the previous part. We create a dictionary to save the scores of the queries and a dictionary for the vectors. We then compute the precision and recall, by comparing our results and the ground truth.

3.3 Results

Table 2 show the statistics of precision and recall compared to the ground truth. We can see that the precision is low for all engines. The engine with the highest precision is `Doc2Vec`. The recall is higher than the precision. The engine `TF-IDF` has a recall equal to 1. The second highest recall is of `Frequencies` with score 0.9. Both `LSI` and `Doc2Vec` have a recall of 0.8.

Engine	**Precision**	**Recall**
Frequencies	0.332	0.9
TD-IDF	0.365	1.0
LSI	0.403	0.8
Doc2Vec	0.417	0.8

Table 2: Statistics of the search engines

4 Visualisation of query results

4.1 Goal and Input parameter

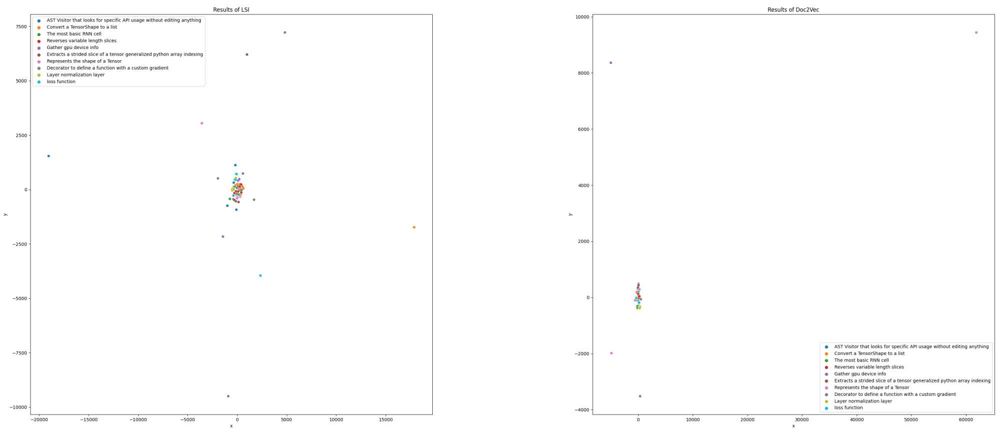
This part of the project consists of visualizing the embedding vectors of the queries and the top 5 answers in a 2D plot. This file takes as argument the ground truth file.

4.2 Description of the code

The first part of the execution is the same as the previous file. After the results are calculated, we plot the TSNE of the embedding vectors, that we retrieved in the explxanation above but we did not use. The plot is straight-forward: we create a dataframe with the information of x and y coordinates and print them of different hues.

4.3 Results

Figure 2 shows the plots of the visualization of the queries.



(a) Results of LSI

(b) Results of Doc2Vec

Figure 2: Visualization of the plots of the queries

A Python code

A.1 Data Extraction

```python
from sys import argv, exit
from ast import *
from os import walk
import pandas as pd


class Visitor(NodeVisitor):
    def __init__(self, file_path, node):
        super().__init__()
        self.file_path = file_path
        self.visit(parse(node))

    def visit_ClassDef(self, node: ClassDef):
        self.generic_visit(node)
        if is_valid_entity(node.name):
            self.append_data(node, "class")

    def visit_FunctionDef(self, node: FunctionDef):
        if is_valid_entity(node.name):
            self.append_data(node, "method" if is_method(node) else "function")

    def append_data(self, node, def_type):
        comment = get_docstring(node)
        comment = comment.split('\n')[0] if comment is not None else ""
        data.append((node.name, self.file_path, node.lineno, def_type, comment))

```

```python

def is_valid_entity(name):
    return name[0] != '_' and name != "main" and "test" not in name.lower()


def is_method(function):
    return function.args and len(function.args.args) > 0 and 'self' in function.args.args[0].arg


def start(directory_path):
    if directory_path[-1] == '/':
        directory_path = directory_path[: -1]
    counter = 0
    for path, _, files in walk(directory_path):
        for file_name in files:
            if file_name.endswith('.py'):
                counter += 1
                file_path = path + '/' + file_name
                with open(file_path) as file:
                    Visitor(file_path, file.read())

    dataframe = pd.DataFrame(data=data, columns=["name", "file", "line", "type", "comment"])
    dataframe.to_csv('res/data.csv', index=False, encoding='utf-8')
    print("files\t    " + str(counter))
    print(dataframe["type"].value_counts())


if len(argv) < 2:
    print("Please give as input the path of the directory to analyze")
    exit(1)
data = []
start(argv[1])
```

A.2 Training of search engines

```python
from datetime import datetime
import string
from os import path
import pandas as pd
import pickle as pkl
from re import finditer
from sys import argv, exit
from collections import defaultdict
from gensim.corpora import Dictionary
from gensim.models.doc2vec import TaggedDocument
from gensim.utils import simple_preprocess
from gensim.models import TfidfModel, LsiModel, Doc2Vec
from gensim.similarities import MatrixSimilarity, SparseMatrixSimilarity


def start(query):
    dataframe = pd.read_csv("res/data.csv").fillna(value="")
    results_dictionary, _ = compute_results(query, dataframe)
    results = pd.DataFrame(data=create_result_dataframe(results_dictionary, dataframe),
```

```
                            columns=['name', "file", "line", "type", "comment", "search"])
    pd.options.display.max_colwidth = 200
    print_results(results)
    results.to_latex('res/search_data.tex', index=False, encoding='utf-8')
    results.to_csv('res/search_data.csv', index=False, encoding='utf-8')


def compute_results(query, dataframe):
    processed_corpus, frequencies, bag_of_words = create_data(dataframe)
    query_to_execute = normalize_query(query)
    results = {
        "FREQ": query_frequency(query_to_execute, bag_of_words, frequencies),
        "TF-IDF": query_tfidf(query_to_execute, bag_of_words, frequencies)
    }
    vectors = dict()
    results["LSI"], vectors["LSI"] = query_lsi(query_to_execute, bag_of_words, frequencies)
    results["Doc2Vec"], vectors["Doc2Vec"] = query_doc2vec(query_to_execute, processed_corpus)
    return results, vectors


def create_data(df):
    tokens = [filter_stopwords(normalize_tokens(handle_camel_case(split_underscore(
        [row["name"]] + split_space(row["comment"]))))) for _, row in df.iterrows()]

    frequency = defaultdict(int)
    for token in tokens:
        for word in token:
            frequency[word] += 1

    processed = [[token for token in text if frequency[token] > 1] for text in tokens]
    dictionary = Dictionary(processed)
    bow = [dictionary.doc2bow(text) for text in processed]

    return processed, dictionary, bow


def split_space(text):
    return text.translate(str.maketrans('', '', string.punctuation)).split(' ') if text != "" else []


def split_underscore(tokens):
    return [word for token in tokens for word in token.split('_')]


def handle_camel_case(tokens):
    words = []
    for token in tokens:
        matches = finditer('.+?(?:(?<=[a-z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])|$)', token)
        words += [m.group(0) for m in matches]
    return words


def normalize_tokens(tokens):
    return [token.lower() for token in tokens]

```

```python

def filter_stopwords(tokens):
    for token in tokens:
        if token in ['test', 'tests', 'main']:
            return []
    return tokens


def normalize_query(query):
    return query.strip().lower().split()


def query_frequency(query, bow, dictionary):
    return filter_results(SparseMatrixSimilarity(bow, num_features=len(dictionary.token2id))[dictionary.doc2bow(query)])


def query_tfidf(query, bow, dictionary):
    model = TfidfModel(bow)
    return filter_results(SparseMatrixSimilarity(model[bow], num_features=len(dictionary.token2id))[model[dictionary.doc2bow(query)]])


def query_lsi(query, bow, dictionary):
    model = LsiModel(bow, id2word=dictionary, num_topics=300)
    vector = model[dictionary.doc2bow(query)]
    result = abs(MatrixSimilarity(model[bow])[vector])
    embedding = [[value for _, value in vector]] + [[value for _, value in model[bow][i]] for i, value in
                                                     sorted(enumerate(result), key=lambda x: x[1], reverse=True)[:5]]
    return filter_results(result), embedding


def filter_results(arrg):
    return [i for i, v in sorted(enumerate(arrg), key=lambda x: x[1], reverse=True)[:5]]


def query_doc2vec(query, corpus):
    model = get_doc2vec_model(get_doc2vec_corpus(corpus))
    vector = model.infer_vector(query)
    similar = model.docvecs.most_similar([vector], topn=5)
    return [index for (index, _) in similar], \
           [list(vector)] + [list(model.infer_vector(corpus[index])) for index, _ in similar]


def get_doc2vec_corpus(corpus):
    return [TaggedDocument(simple_preprocess(' '.join(element)), [index])
            for index, element in enumerate(corpus)]


def get_doc2vec_model(corpus):
    return pkl.load(open('res/doc2vec.pkl', "rb")) if path.exists('res/doc2vec.pkl') else create_doc2vec_model(corpus)


def create_doc2vec_model(corpus):
```

```
    model = Doc2Vec(vector_size=300, min_count=2, epochs=77)
    model.build_vocab(corpus)
    model.train(corpus, total_examples=model.corpus_count, epochs=model.epochs)
    pkl.dump(model, open('res/doc2vec.pkl', "wb"), protocol=pkl.HIGHEST_PROTOCOL)
    return model


def create_result_dataframe(queries_dictionary, df):
    for key, values in queries_dictionary.items():
        for index in sorted(values):
            row = df.iloc[index]
            yield [row["name"], row["file"], row["line"], row["type"], row["comment"], key]


def print_results(df):
    grouped = df.groupby(['search'])
    for key, item in grouped:
        print(grouped.get_group(key), "\n\n")


if len(argv) < 2:
    print("Please give as input the query")
    exit(1)

begin_time = datetime.now()
start(argv[1])
print(datetime.now() - begin_time)
```

A.3 Evaluation of search engines and Visualisation of query results

```
import itertools
from datetime import datetime

import string
import pandas as pd
from os import path
import pickle as pkl
import seaborn as sns
from re import finditer
from sys import argv, exit
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
from collections import defaultdict
from gensim.corpora import Dictionary
from gensim.models.doc2vec import TaggedDocument
from gensim.utils import simple_preprocess
from gensim.models import TfidfModel, LsiModel, Doc2Vec
from gensim.similarities import MatrixSimilarity, SparseMatrixSimilarity

##################
def get_results(query, dataframe):
    results_dictionary, vectors = compute_results(query, dataframe)
    return pd.DataFrame(data=create_result_dataframe(results_dictionary, dataframe),
                        columns=['name', "file", "line", "type", "comment", "search"]),
                            vectors
```

```


def compute_results(query, dataframe):
    processed_corpus, frequencies, bag_of_words = create_data(dataframe)
    query_to_execute = normalize_query(query)
    results = {
        "FREQ": filter_results(query_frequency(query_to_execute, bag_of_words, frequencies)),
        "TF-IDF": filter_results(query_tfidf(query_to_execute, bag_of_words, frequencies))
    }
    vectors = dict()
    results["LSI"], vectors["LSI"] = query_lsi(query_to_execute, bag_of_words, frequencies)
    results["Doc2Vec"], vectors["Doc2Vec"] = query_doc2vec(query_to_execute, processed_corpus)
    return results, vectors


def create_data(df):
    tokens = [filter_stopwords(normalize_tokens(handle_camel_case(split_underscore(
        [row["name"]] + split_space(row["comment"]))))) for _, row in df.iterrows()]

    frequency = defaultdict(int)
    for token in tokens:
        for word in token:
            frequency[word] += 1

    processed = [[token for token in text if frequency[token] > 1] for text in tokens]
    dictionary = Dictionary(processed)
    bow = [dictionary.doc2bow(text) for text in processed]

    return processed, dictionary, bow


def split_space(text):
    return text.translate(str.maketrans('', '', string.punctuation)).split(' ') if text != "" else []


def split_underscore(tokens):
    return [word for token in tokens for word in token.split('_')]


def handle_camel_case(tokens):
    words = []
    for token in tokens:
        matches = finditer('.+?(?:(?<=[a-z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])|$)', token)
        words += [m.group(0) for m in matches]
    return words


def normalize_tokens(tokens):
    return [token.lower() for token in tokens]


def filter_stopwords(tokens):
    for token in tokens:
        if token in ['test', 'tests', 'main']:
            return []
```

```python
    return tokens


def normalize_query(query):
    return query.strip().lower().split()


def query_frequency(query, bow, dictionary):
    return SparseMatrixSimilarity(bow, num_features=len(dictionary.token2id))[dictionary.doc2bow(query)]


def query_tfidf(query, bow, dictionary):
    model = TfidfModel(bow)
    return SparseMatrixSimilarity(model[bow], num_features=len(dictionary.token2id))[model[dictionary.doc2bow(query)]]


def query_lsi(query, bow, dictionary):
    model = LsiModel(bow, id2word=dictionary, num_topics=300)
    vector = model[dictionary.doc2bow(query)]
    result = abs(MatrixSimilarity(model[bow])[vector])
    embedding = [[value for _, value in vector]] + [[value for _, value in model[bow][i]] for i, value in
                                                       sorted(enumerate(result), key=lambda x: x[1], reverse=True)[:5]]
    return filter_results(result), embedding


def filter_results(arrg):
    return [i for i, v in sorted(enumerate(arrg), key=lambda x: x[1], reverse=True)[:5]]


def query_doc2vec(query, corpus):
    model = get_doc2vec_model(get_doc2vec_corpus(corpus))
    vector = model.infer_vector(query)
    similar = model.docvecs.most_similar([vector], topn=5)
    return [index for (index, _) in similar], \
           [list(vector)] + [list(model.infer_vector(corpus[index])) for index, _ in similar]


def get_doc2vec_corpus(corpus):
    return [TaggedDocument(simple_preprocess(' '.join(element)), [index])
            for index, element in enumerate(corpus)]


def get_doc2vec_model(corpus):
    return pkl.load(open('res/doc2vec.pkl', "rb")) if path.exists('res/doc2vec.pkl') else create_doc2vec_model(corpus)


def create_doc2vec_model(corpus):
    model = Doc2Vec(vector_size=300, min_count=2, epochs=77)
    model.build_vocab(corpus)
    model.train(corpus, total_examples=model.corpus_count, epochs=model.epochs)
    pkl.dump(model, open('res/doc2vec.pkl', "wb"), protocol=pkl.HIGHEST_PROTOCOL)
    return model
```

```python


def create_result_dataframe(queries_dictionary, df):
    for key, values in queries_dictionary.items():
        for index in sorted(values):
            row = df.iloc[index]
            yield [row["name"], row["file"], row["line"], row["type"], row["comment"], key]


####################################

class Truth:
    def __init__(self, query, name, path):
        self.name = name
        self.path = path
        self.query = query.lower()


class Stat:
    def __init__(self, precisions, recalls):
        self.precisions = precisions
        self.recalls = recalls


def start(path_ground_truth):
    dataframe = pd.read_csv("res/data.csv").fillna(value="")
    ground_truth, queries = parse_ground_truth(path_ground_truth)
    scores, vectors = compute_precision_recall(ground_truth, dataframe)
    plot_vectors(compute_tsne(vectors), queries)
    print_scores(scores)


def parse_ground_truth(path_ground_truth):
    classes, queries = [], []
    for entry in open(path_ground_truth, "r").read().split("\n\n"):
        data = entry.split("\n")
        classes.append(Truth(data[0], data[1], data[2]))
        queries.append(data[0])
    return classes, queries


def compute_precision_recall(ground_truth, dataframe):
    scores = {"FREQ": [], "TF-IDF": [], "LSI": [], "Doc2Vec": []}
    vectors = {"LSI": [], "Doc2Vec": []}
    for entry in ground_truth:
        results, vectors_i = get_results(entry.query, dataframe)
        vectors["LSI"] += vectors_i["LSI"]
        vectors["Doc2Vec"] += vectors_i["Doc2Vec"]
        for query_type in ["FREQ", "TF-IDF", "LSI", "Doc2Vec"]:
            precision = compute_precision(entry, query_type, results)
            scores[query_type].append(Stat(precision, compute_recall(precision)))
    return scores, vectors


def compute_precision(truth, search_type, dataframe):
    precision, counter = 0, 0
    for _, row in dataframe[dataframe['search'] == search_type].iterrows():
```

```
        if row["name"] == truth.name and row["file"] == truth.path:
            return 1 / (counter + 1)
        counter += 1
    return precision


def compute_recall(precision):
    return 1 if precision > 0 else 0


def compute_tsne(dictionary):
    results = {}
    for key, values in dictionary.items():
        tsne = TSNE(n_components=2, verbose=1, perplexity=2, n_iter=3000)
        results[key] = tsne.fit_transform(values)
    return results


def plot_vectors(dictionary, queries):
    for key, values in dictionary.items():
        dataframe = pd.DataFrame()
        dataframe['x'] = values[:, 0]
        dataframe['y'] = values[:, 1]
        plt.figure(figsize=(16, 16))
        plt.title("Results of " + key)

        sns_plot = sns.scatterplot(
            x="x",
            y="y",
            hue=queries + list(itertools.chain.from_iterable([query] * 5 for query in queries
                )),
            data=dataframe,
            legend="full",
            alpha=1.0
        )
        sns_plot.get_figure().savefig("res/plot_" + key.lower())


def print_scores(scores):
    print("##### PRINT #####")
    for key, values in scores.items():
        print(key)
        precision, recall = compute_mean(values)
        print("\tprecision:\t" + precision)
        print("\trecall:\t\t" + recall)


def compute_mean(stats):
    precision, recall, counter = 0, 0, 0
    for stat in stats:
        precision += stat.precisions
        recall += stat.recalls
        counter += 1
    return str(precision / counter), str(recall / counter)


if len(argv) < 1:
```

```
    print("Please give as input ground truth file")
    exit(1)


begin_time = datetime.now()
start(argv[1])
print(datetime.now() - begin_time)
```

B Bash Code

```
#!/bin/bash

python3 src/extract_data.py $1
python3 src/search_data.py $2
python3 src/prec_recall.py res/data.csv res/ground-truth.txt
```