Project 2: Multi-source code search

Ardigò Susanna

December 11, 2020

https://github.com/SusyPinkBash/multi_source_code_search

Contents

1 Data Extraction **2**

1.1 Goal and Input parameter . . . 2

1.2 Description of the code . . . 2

1.3 Results . . . 2

2 Training of search engines **3**

2.1 Goal and Input parameter . . . 3

2.2 Description of the code . . . 3

2.3 Results . . . 3

3 Evaluation of search engines **4**

3.1 Goal and Input parameter . . . 4

3.2 Description of the code . . . 4

3.3 Results . . . 5

4 Visualisation of query results **5**

4.1 Goal and Input parameter . . . 5

4.2 Description of the code . . . 5

4.3 Results . . . 5

4.3.1 LSI . . . 6

4.3.2 Doc2Vec . . . 6

A Python code **7**

A.1 Data Extraction . . . 7

A.2 Training of search engines . . . 8

A.3 Evaluation of search engines and Visualisation of query results . . . 12

B Bash Code **18**

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 if the node is a method by checking 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 we 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 Tensorflow directory. The results can be found in the file `res/data.csv`.

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 which will be fed to the four search engines.

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 normalizes the query that the second part needs to produce the results.
The first part of function `create_data(dataframe)` extracts the names and comments of the data extracted in the first part. 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. At the end of the creation, we save the corpus, dictionary and bag of words in external files to then load them in future runs. 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 create a vector infering it from the query, we create the similarity and take only the top 5 scores and the embedding vectors.
We save the trained models in external pickle files to load then load them in the next runs. This improves the running time of the function.
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:

'AST Visitor that looks for specific API usage without editing anything' .

The correct document is `PastaAnalyzeVisitor` with path `../tensorflow/tensorflow/tools/compatibility/ast_edits.py`.
Figure 1 show the result of the given query.
As we can see in the image all search engine find the correct result. Frequency, TF-IDF and LSI find it as the first result. Doc2Vec finds the correct result as second result. The first result is a class in the same file but named `APIChangeSpec` and with a different comment. This result is found only by this search engine. Doc2Vec has different results compared to the other search engines. Frequency and TF-IDF have the most similar results.

	name	file	line	type	comment	search
1	PastaAnalyzeVisitor	../tensorflow/tensorflow/tools/compatibility/ast_edits.py	815	class	AST Visitor that looks for specific API usage without editing an…	FREQ
2	CompatV1ImportRepla…	../tensorflow/tensorflow/tools/compatibility/tf_upgrade_v2.py	72	class	AST Visitor that replaces `import tensorflow.compat.v1 as tf`.	FREQ
3	CleanCopier	../tensorflow/tensorflow/python/autograph/pyct/ast_util.py	30	class	NodeTransformer-like visitor that copies an AST.	FREQ
4	FunctionVisitor	../tensorflow/tensorflow/python/autograph/pyct/static_analysis/type_infer…	394	class	AST visitor that applies type inference to each function separat…	FREQ
5	track_usage	../tensorflow/tensorflow/python/platform/analytics.py	21	function	No usage tracking for external library.	FREQ
6	PastaAnalyzeVisitor	../tensorflow/tensorflow/tools/compatibility/ast_edits.py	815	class	AST Visitor that looks for specific API usage without editing an…	TF-IDF
7	PublicAPIVisitor	../tensorflow/tensorflow/tools/common/public_api.py	29	class	Visitor to use with `traverse` to visit exactly the public TF AP…	TF-IDF
8	CleanCopier	../tensorflow/tensorflow/python/autograph/pyct/ast_util.py	30	class	NodeTransformer-like visitor that copies an AST.	TF-IDF
9	FunctionVisitor	../tensorflow/tensorflow/python/autograph/pyct/static_analysis/type_infer…	394	class	AST visitor that applies type inference to each function separat…	TF-IDF
10	track_usage	../tensorflow/tensorflow/python/platform/analytics.py	21	function	No usage tracking for external library.	TF-IDF
11	PastaAnalyzeVisitor	../tensorflow/tensorflow/tools/compatibility/ast_edits.py	815	class	AST Visitor that looks for specific API usage without editing an…	LSI
12	CountingVisitor	../tensorflow/tensorflow/python/autograph/pyct/cfg_test.py	28	class	<null>	LSI
13	matches	../tensorflow/tensorflow/python/autograph/pyct/ast_util.py	214	function	Basic pattern matcher for AST.	LSI
14	concentration	../tensorflow/tensorflow/python/ops/distributions/dirichlet.py	213	method	Concentration parameter; expected counts for that coordinate.	LSI
15	ProfileOptionBuilder	../tensorflow/tensorflow/python/profiler/option_builder.py	27	class	Option Builder for Profiling API.	LSI
16	APIChangeSpec	../tensorflow/tensorflow/tools/compatibility/ast_edits.py	192	class	This class defines the transformations that need to happen.	Doc2Vec
17	PastaAnalyzeVisitor	../tensorflow/tensorflow/tools/compatibility/ast_edits.py	815	class	AST Visitor that looks for specific API usage without editing an…	Doc2Vec
18	Trackable	../tensorflow/tensorflow/python/training/tracking/base.py	537	class	Base class for `Trackable` objects without automatic dependencie…	Doc2Vec
19	check_dtype	../tensorflow/tensorflow/python/ops/linalg/linear_operator_util.py	139	function	Check that arg.dtype == self.dtype.	Doc2Vec
20	ProfileOptionBuilder	../tensorflow/tensorflow/python/profiler/option_builder.py	27	class	Option Builder for Profiling API.	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 unique ground truth. We can see that the precision is low for all engines. The engine with the highest precision is `LSI`, which is the only score higher than 0.4. The second highest precision is `TF-IDF`, followed by `Frequencies` and then **Doc2Vec**. We can say that almost all search engine have a similar precision.
The recall is higher than the precision. The `TF-IDF` engine has a recall equal to 1, which means that for each query the search engine has found the correct result in the top 5. 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	0.323	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 use the embedding vectors, that we retrieved in the explanation above but we did not use. For vector we apply TSNE to produce 2D vectors composed of queries and the top 4 results. The plot is straight-forward: we create a dataframe with the information of x and y coordinates and print them of different hues. We use the library `seaborn` to create the charts and we then save them on disk.

4.3 Results

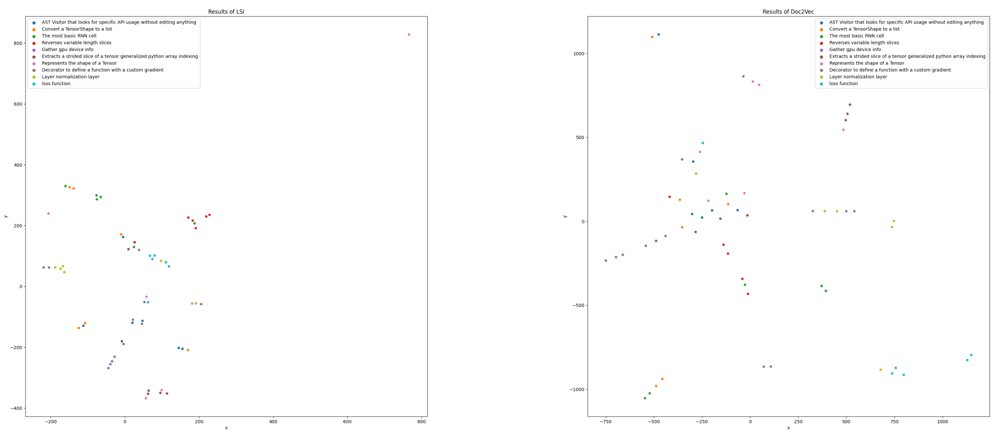
Figure 2 shows the plots of the visualization of the queries. At first we notice that the LSI scatterplot tends to be more a bit compact, while the Doc2Vec scatterplot is more sparse. The optimal solution is to have defined clusters for each query. This does not happen in any of the two images.

4.3.1 LSI

Analyzing the plot of LSI, shown in figure 2a, we can see that some results of the queries tend to stay close, but not completely. There are some clusters that are close. In most cases the cluster have different queries. The most well defined cluster is *'Gather gpu device info'*, colored in purple. The sparseness of the data reflects the low precision.

4.3.2 Doc2Vec

Analyzing the plot of Doc2Vec, shown in figure 2b, we can see that few results of the queries tend to stay close. In this image there are some cluster with more queries. The most well defined cluster is *'Gather gpu device info'*, as in the previous plot. The sparseness of the data reflects the low precision, which is lower than the LSI. This is reflected in the plot.



(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 = clean_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))


def clean_file_path(path):
    directories = path.split('/')
    return '../' + '/'.join(directories[directories.index('tensorflow'):])


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())
```

```python

    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 = get_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


```

```python
def get_data(df):
    return load_data_files() if exists_data_files() else create_data(df)


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

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

    save_data(corpus, 'corpus')
    save_data(dictionary, 'dictionary')
    save_data(bow, 'bow')
    return corpus, dictionary, bow


def exists_data_files():
    return exists_file('corpus') and exists_file('dictionary') and exists_file('bow')


def exists_file(name):
    return path.exists('res/' + name + '.pkl')


def load_data_files():
    return load_file('corpus'), load_file('dictionary'), load_file('bow')


def save_data(data, name):
    pkl.dump(data, open('res/' + name + '.pkl', "wb"), protocol=pkl.HIGHEST_PROTOCOL)


def load_file(name):
    return pkl.load(open('res/' + name + '.pkl', "rb"))


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 []
    return tokens


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


def save_model(model, name):
    save_data(model, 'model_' + name)


def query_frequency(query, bow, dictionary):
    return filter_results(get_freq_model(bow, dictionary)[dictionary.doc2bow(query)])


def get_freq_model(bow, dictionary):
    return load_file('model_freq') if exists_file('model_freq') else create_freq_model(bow,
        dictionary)


def create_freq_model(bow, dictionary):
    model = SparseMatrixSimilarity(bow, num_features=len(dictionary.token2id))
    save_model(model, 'freq')
    return model


def query_tfidf(query, bow, dictionary):
    model = get_tfidf_model(bow)
    matrix = get_tfidf_matrix(model, bow, dictionary)
    return filter_results(matrix[model[dictionary.doc2bow(query)]])


def get_tfidf_model(bow):
    return load_file('model_tfidf') if exists_file('model_tfidf') else create_tfidf_model(bow
        )


def create_tfidf_model(bow):
    model = TfidfModel(bow)
    save_model(model, 'tfidf')
    return model


def get_tfidf_matrix(model, bow, dictionary):
    return load_file('matrix_tfidf') if exists_file('matrix_tfidf') else create_tfidf_matrix(
        model, bow, dictionary)
```

```python
def create_tfidf_matrix(model, bow, dictionary):
    matrix = SparseMatrixSimilarity(model[bow], num_features=len(dictionary.token2id))
    save_data(matrix, 'matrix_tfidf')
    return model


def query_lsi(query, bow, dictionary):
    model = get_lsi_model(bow, dictionary)
    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 get_lsi_model(bow, dictionary):
    return load_file('model_lsi') if exists_file('model_lsi') else create_lsi_model(bow, dictionary)


def create_lsi_model(bow, dictionary):
    model = LsiModel(bow, id2word=dictionary, num_topics=300)
    save_model(model, 'lsi')
    return model


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 load_file('model_doc2vec') if exists_file('model_doc2vec') 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)
    save_model(model, 'doc2vec')
```

```python
    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)

start(argv[1])
```

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

```python
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 = get_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 get_data(df):
    return load_data_files() if exists_data_files() else create_data(df)


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

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

    save_data(corpus, 'corpus')
    save_data(dictionary, 'dictionary')
    save_data(bow, 'bow')
    return corpus, dictionary, bow


def exists_data_files():
    return exists_file('corpus') and exists_file('dictionary') and exists_file('bow')


def exists_file(name):
    return path.exists('res/' + name + '.pkl')


def load_data_files():
    return load_file('corpus'), load_file('dictionary'), load_file('bow')


def save_data(data, name):
    pkl.dump(data, open('res/' + name + '.pkl', "wb"), protocol=pkl.HIGHEST_PROTOCOL)


def load_file(name):
    return pkl.load(open('res/' + name + '.pkl', "rb"))


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 []
    return tokens


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


def save_model(model, name):
    save_data(model, 'model_' + name)


def query_frequency(query, bow, dictionary):
    return filter_results(get_freq_model(bow, dictionary)[dictionary.doc2bow(query)])


def get_freq_model(bow, dictionary):
    return load_file('model_freq') if exists_file('model_freq') else create_freq_model(bow,
        dictionary)


def create_freq_model(bow, dictionary):
    model = SparseMatrixSimilarity(bow, num_features=len(dictionary.token2id))
    save_model(model, 'freq')
    return model


def query_tfidf(query, bow, dictionary):
    model = get_tfidf_model(bow)
    matrix = get_tfidf_matrix(model, bow, dictionary)
    return filter_results(matrix[model[dictionary.doc2bow(query)]])


def get_tfidf_model(bow):
    return load_file('model_tfidf') if exists_file('model_tfidf') else create_tfidf_model(bow
        )

```

```

def create_tfidf_model(bow):
    model = TfidfModel(bow)
    save_model(model, 'tfidf')
    return model


def get_tfidf_matrix(model, bow, dictionary):
    return load_file('matrix_tfidf') if exists_file('matrix_tfidf') else create_tfidf_matrix(
        model, bow, dictionary)


def create_tfidf_matrix(model, bow, dictionary):
    matrix = SparseMatrixSimilarity(model[bow], num_features=len(dictionary.token2id))
    save_data(matrix, 'matrix_tfidf')
    return model


def query_lsi(query, bow, dictionary):
    model = get_lsi_model(bow, dictionary)
    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 get_lsi_model(bow, dictionary):
    return load_file('model_lsi') if exists_file('model_lsi') else create_lsi_model(bow,
        dictionary)


def create_lsi_model(bow, dictionary):
    model = LsiModel(bow, id2word=dictionary, num_topics=300)
    save_model(model, 'lsi')
    return model


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 load_file('model_doc2vec') if exists_file('model_doc2vec') 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)
    save_model(model, 'doc2vec')
    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]


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

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):
    counter = 0
    for _, row in dataframe[dataframe['search'] == search_type].iterrows():
        counter += 1
        if row["name"] == truth.name and row["file"] == truth.path:
            return 1 / counter
    return 0


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 + "\n\trecall:\t\t" + recall)

```

```python
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)


start(argv[1])
```

B Bash Code

```bash
#!/bin/bash

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