Eötvös Loránd Tudományegyetem
Informatikai Kar
Data Science and Energeeing Department

Decision making supporting with essay score prediction

Dr. Tamás Horváth
Head of Department

Dávid Szabó
Programtervező Informatikus BSc

Budapest, 2019

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

SZAKDOLGOZAT-TÉMA BEJELENTŐ

Név: Szabó Dávid Neptun kód: JLF6V4

Tagozat: nappali (**nappali** vagy **esti**) Szak: **programtervező informatikus BSc**

Témavezető neve: Dr. Horváth Tamás

munkahelyének neve és címe: 1117 Budapest, Pázmány Péter sétány 1/C

beosztása és iskolai végzettsége: Tanszékvezető

A dolgozat címe: **Decision making supporting with essay score prediction**

A dolgozat témája:

As widely known, in the modern social system, where the education is available for the mass, a lot of overloaded teachers suffer from the huge number of essays and tests, which have to be corrected. Very often, the students have to write the tests about the same topic so the scoring process can be very boring and monotonous because they make the same statements and the same errors again and again. This trend will increase with the spread of online learning systems, where fortunately the essays are in digital format.

In this paper, I propose a method and a framework on essay score prediction in order to support the decision making of teachers who are expected to correct large quantities of essays on the same topic. The proposed method will be implemented in Python due to the fact, for Python, there are many available libraries for supporting development especially in the field of Data Science.

As almost every Data Science process, this solution also can be divided into three important and individual steps. The first step is preprocessing the data, and transforming it into a proper format. The second step is making predictions on the target values based on the model. Finally, the third step is the evaluation, which indicates how good the proposed model and that's hyperparameters are on a certain task. If the results are satisfactory, a good recommender system can be made by the process.

Since this method has to find the essays with similar meanings for predicting the scores of ones by others, the model has to be able to represent the text in a way that the semantic meaning of essays will be comparable easily. For this reason, the method requires a suitable natural language model, and for time-efficiency, it uses estimating methods to measure similarity.

A témavezetést vállalom:

.....
(a témavezető aláírása)

Kérem a szakdolgozat témájának jóváhagyását.

Budapest, 20…….

.....
(a hallgató aláírása)

A szakdolgozat-témát az Informatikai Kar jóváhagyta.

Budapest, 20……

.....
(a témát engedélyező tanszék vezetője)

Contents

Acknowledgement

I would like to express my special thanks of gratitude to my supervisor Dr. Tamás Horváth for his great assistance and for the opportunity to join in his research with Tsegaye Misikir, who also helped me through my work.

Finally, I must express my very profound gratitude to my parents and to my fiancée Dóra Kovács for providing me with continuous encouragement and support throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Thank you.

Dávid Szabó

1 Introduction

2 User documentation

2.1 Problem

In Data Science, working with textual dataset is a highly challenging task as the characterisation of a text or comparison of documents can be very difficult. However, there are a large number of real-life problems which are significantly related to or depend on textual data, such as automatic essay score evaluation, spam filtering and sentiment analysis. The problem is the complexity and the lack of a profound understanding of natural languages. In computer science, we like to work with quantitative and numerical data because we have mathematical and numerical tools to make computations easily with them. Unfortunately, so far we can not express the meaning of a document or even a sentence with numbers without a great loss. However, there is some method which can be used to represent documents as numerical data to solve some problems well enough to apply them in real life. In this essay, I propose a method for scoring texts by the similarity to other texts annotated by humans. This is a so-called supervised learning task. With other words, the model tries to learn a function which maps the input (text) to an output (score) based on example input-output pairs, called training data set.

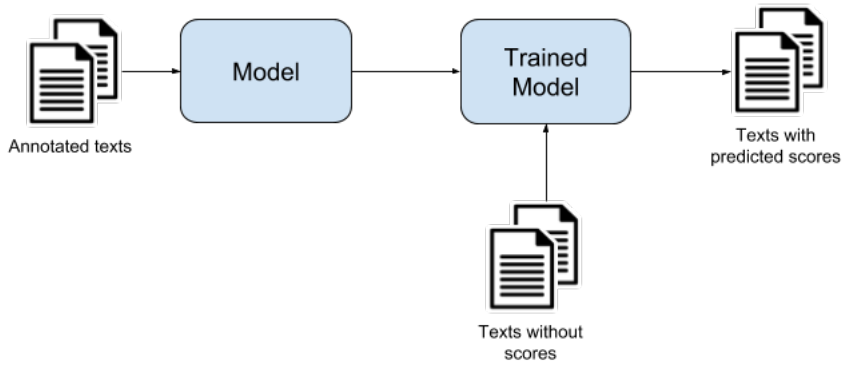


Figure 1: Process flow from perspective of model user

Though this problem specification fit on a lot of different tasks, I will present the proposed model on the following two datasets (one for a regression task, and one for a classification task):

Automatic essay score evaluation

The texts are short essays which are written about a certain topic, and the scores are the grades which are given by a teacher. The goal is to predict the grade of the uncorrected essays to support the decision making of the corrector. To determine the performance of the model, the annotated dataset will be divided into a training

dataset to train the model and a test dataset to measure the performance of the trained model. As a grade is a quantitative and ordinal value, this is a regression task. The dataset is provided by the Hewlett Foundation at Kaggle[1].

Spam filtering

The texts are short tweets which are classified as aggressive or not, thereby the scores are 1 or 0, respectively. The goal is to determine whether a tweet is aggressive or not. To assess the performance of the model, the annotated dataset will be divided into a training dataset to train the model and a test dataset to measure the performance of the trained model. As the score is a binary, nominal value, this is a classification task. The dataset is provided by user DataTurks at Kaggle[2].

2.2 Solution

As almost every Data Science process, this solution also can be divided into three important, sequential and individual steps. The first step is preprocessing the data, and transforming it into a proper format. The second step is making predictions on the target values based on the model. Finally, the third step is the evaluation, which indicates how good the proposed model and that's hyper-parameters are on a certain task. If the results are satisfactory, a good recommender system can be made by the process. To make these steps easier, I have created a Jupyter Notebook including preprocessing, modelling and evaluation actions to generate a report about the performance of this algorithm on a certain dataset. As notebooks can be edited easily, further modifications and adjustments can be performed simply by a Data Scientist wanting experiment a supplementation of the model.

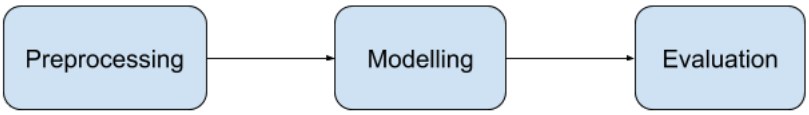


Figure 2: Main steps of a data science project

2.2.1 Preprocessing step

We can regard preprocessing as a noise filtering or dimensionality reduction step as it produces simpler representation by removing special characters, numbers from the text and converting characters to lowercase. To make this part simpler to do a class called "TextTransformation" was created, which can help to apply the most

[1] https://www.kaggle.com/c/asap-sas
[2] https://www.kaggle.com/dataturks/dataset-for-detection-of-cybertrolls

frequent text transformations on the documents to improve the performance of further operations. By default, in this step, the text will be converted to lowercase furthermore numbers, special characters and stopwords will be removed.

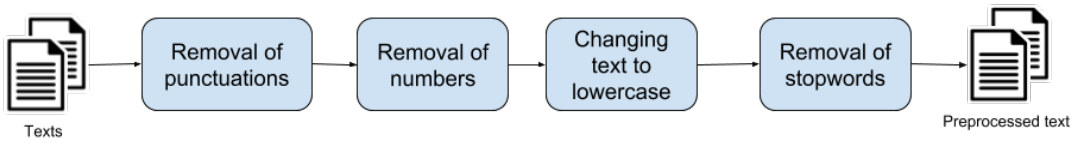


Figure 3: A possible pipeline for text preprocessing

2.2.2 Modelling step

First and foremost, we need to represent texts such that we could compare them to each other easily. In this phase, we implemented two different approaches, the first one using n-grams and the other one using Word2Vec vectors. For n-gram representation, each text will be converted into a Bag of Words(BoW) by extracting n-grams from them, where n-gram is a contiguous sequence of n characters from the given text. In this way, we have a set of strings instead of a raw text to represent an instance. For Word2Vec vectors, each word from each text will be mapped into a high dimensional vector space with a trained word embedding function so that for each text we will have a set of high dimensional vectors.

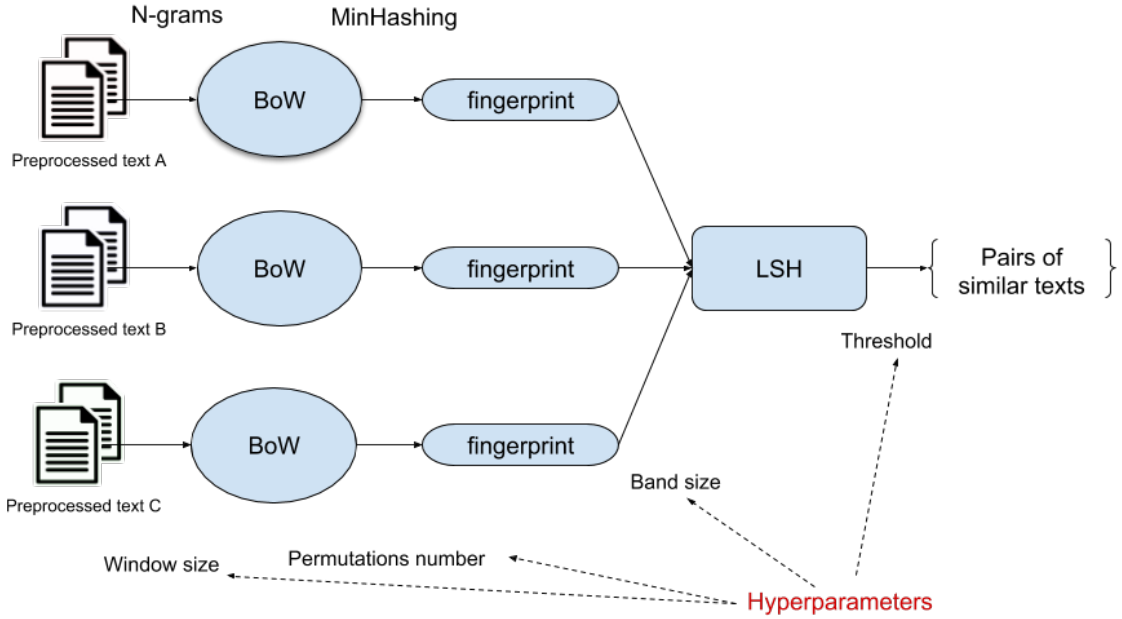


Figure 4: Simplified overview of the proposed model

Presumably, similar texts will have similar scores, consequently, similar set representations will have approximately similar scores as well. Jaccard Index is one of

the most popular metrics expressing the similarity of two sets, however, unfortunately, if we want to find the most similar sets we need to do $\frac{n^2}{2}$ comparison, which is unfeasible in case of large databases. So instead of computing the similarity for all pairs, the algorithm represents all set of n-grams or set of vectors with a fingerprint of size n by MinHashing them with a permutation number n. Note: the hashing technique differs in the two different text representation. Based on those fingerprints, LSH (Locality-sensitive hashing) can recommend pairs which are likely to similar to each other, and the time complexity of that is just n. In this way, we are able to reduce the number of comparisons. Next, we compute the real Jaccard index of the recommended pairs to filter out the false positives whose similarity value are not larger than a certain threshold. Finally, we can predict the score of a given text by calculating the average of the scores of similar texts from the annotated datasets.

For further information about the above-mentioned methods you can check the developer documentation, however, here is some heuristic which may help to find an optimal hyper-parameter set:

Window size: (the length of n-grams, positive integer) If we pick n too small, all documents will appear to be similar to each other, however, if we pick it too large, even similar documents will have a totally different representation. The aim is to choose n large enough to have the probability of an n-gram occurring in a text low but not too low. As a result, to choose an optimal window size, one should consider the cardinality of the character set and the average length of texts in your dataset.

Permutation Number: (hyper-parameter of MinHash, positive integer) It is mathematically proved that if permutation number tends to infinity, the Jaccard Index of two fingerprints generated from two sets by MinHashing tends to the Jaccard Index of the two sets. However, with a low permutation number, we also can reach a satisfactory performance. To choose an optimal permutation number, one should strike a balance between running time and the probability of good approximation.

Band size: (hyper-parameter of LSH, positive integer) It is important to mention that, permutation number must be divisible by band size. To generate recommended pairs, LSH splits each fingerprint into slices of the length of band size and maps these slices into buckets. Two objects will be suggested to be similar if at least one of their slices appears in the same bucket. Therefore, if band size smaller, there will be more matches and, naturally, more false positives, however, if it is too large, a lot of similar items will not be found.

Thus, to find an optimal band size, one should strike a balance between too more and too few matches.

Threshold (floating-point number range between $[0; 1]$): Having got the recommended pairs by LSH, we probably have some false positives so before we concern a pair to be similar, we compute the Jaccard Index for their fingerprint and regard them as similar only if the index is larger than the threshold. It depends on the task and the quality of fingerprints that how similar items we want to regard as similar.

2.2.3 Evaluation step

The purpose of this step is measuring how well a model, with a given parameter set, can predict the score of a previously unseen input, with other words how well the model can generalize from the training data to the unseen data. Thus, to evaluate the performance, we usually need to have training and test dataset so we can train the model with the former and evaluate with the latter. One solution could be simply splitting the dataset into random training and test subsets. However, the performance of the model can depend highly on the quality of the random split.

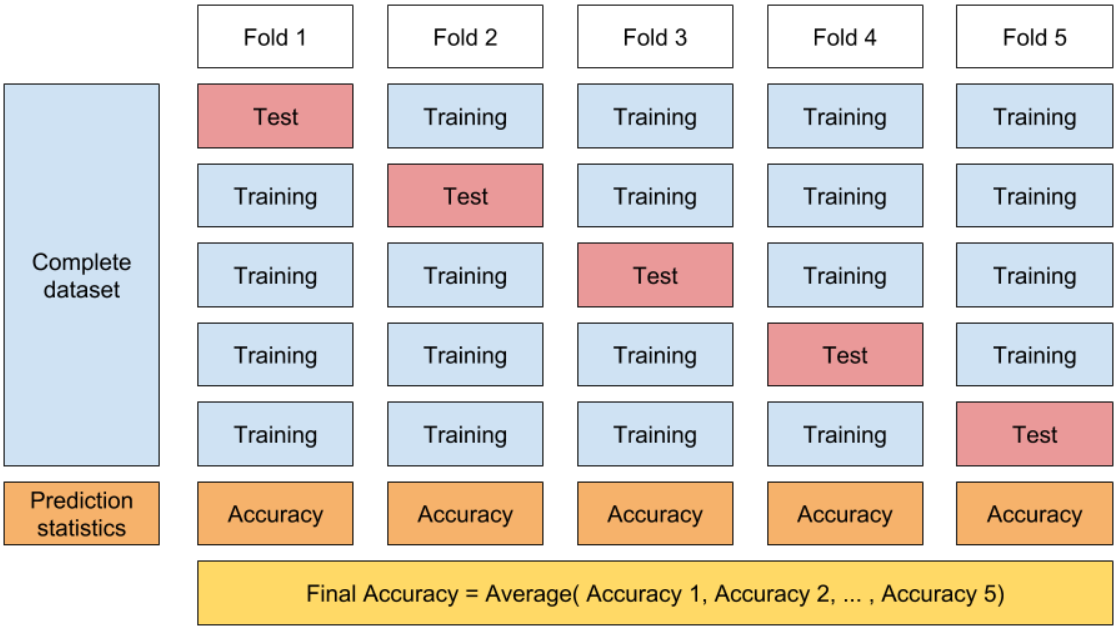


Figure 5: Visualization of cross validation

An improved technique is cross-validation, also called rotation estimation, which splits the annotated dataset into n equal fragments, train the model once with each combination of n 1 fragments and test the trained model with the remaining one fragment. The final accuracy or performance of a certain model will be the average performance of the n evaluation. This method reduces the overfitting as the model

will use most of the data for validation and also reduce the underfitting as the majority of the data will be used as the training set also. Parenthetically, overfitting means that the model performs very well on the training dataset but very poorly on the test dataset, in contrast to underfitting, when the model performs poorly both of the training and the test dataset.

As I previously mentioned, I will present this model on two different dataset for two different tasks, one for regression and one for classification, for both of them we need to use different kinds of metrics to express their performance.

For classification by default, the evaluation step will use the accuracy metric, which is defined as the percentage of correct predictions. It is a sufficient basic metric, though, if for example, a dataset consists significantly more class from one than from the other, the result can be misleading. F1 score is more general and more meaningful in those cases.

For regression problems, we rather like to express the error of the model instead of the accuracy, which is hardly interpretable in this case. If the error is smaller, the model is better. The default metric what the evaluation step use to indicate the error is Mean Absolute Error (MAE). However, for certain tasks, Root Mean Square Error (RMSE), which penalty the larger deviations better than the smaller, is more meaningful and expressive and would be a better choice. For cross-validation we sometimes use these metrics with a negative sign so if the negative error is larger, the model is better.

Please consult the developer documentation for more details about the implemented metrics for classification and regression.

2.3 Requirements

The method is implemented using Python 3, the necessary Python packages can be installed with pip3 (The Python Package Installer) and the user interface is a Jupyter Notebook. As Python is a cross-platform language and Jupyter Notebook supports all popular operating systems, we can regard this code as platform independent.

2.3.1 Software requirements

Programs required to be installed:

A suitable browser (Chrome, Firefox, Safari etc.)

Python 3.5 or above

pip3

2.3.2 Package requirements

After the above-listed programs have been installed, the following Python packages are also necessary to be installed by pip3 (console: pip install "package name"):

jupyter

numpy

pandas

networkx

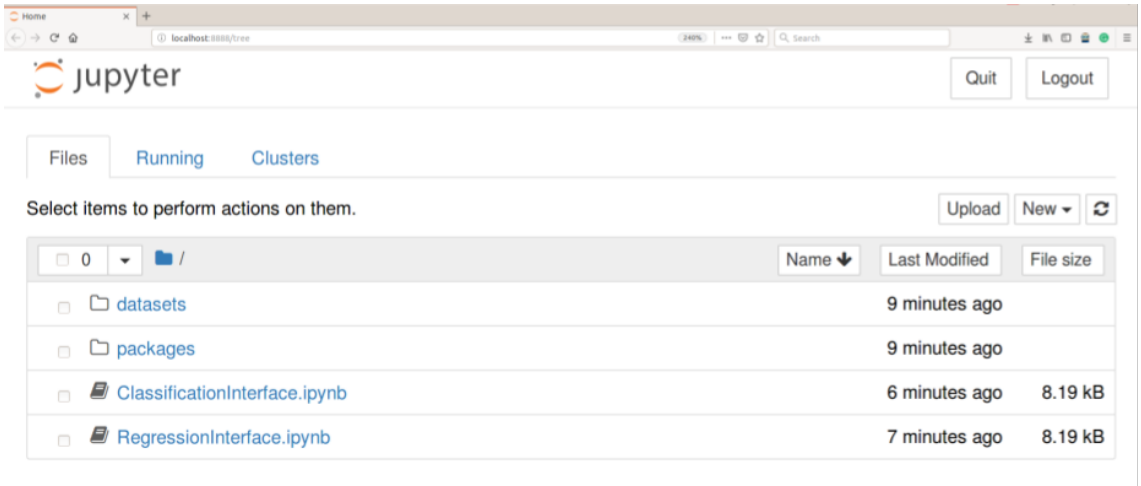
unittest

matplotlib

A virtual Conda environment with the above-listed packages can also be sufficient. Running

2.4 Running

In this section, you can find a step by step guide for starting a Jupyter Notebook Server on your computer in the local environment and analysing a dataset with the proper notebook.



Steps:

1. Check if your system meets with all requirements, listed in the previous section. If not, please install all required software and packages or if it is not possible you can run it in the clouds or in a remote server.

2. Download SemanticGraph.zip and unzip it into your working folder.

3. Open a terminal (or a command line) in your working folder, and run the following command to start a local server at 8888 port. You can change the port number to another if 8888 is already reserved. If you want to run your notebook server in a remote computer and want to use it in your personal computer you need to do port forwarding with SSH with this port number also. jupyter notebook –port=8888

4. Having started a notebook server, the program will print a link whereby you can access to the hosted jupyter notebook application running either on your local computer or a remote server. With this Dashboard, you can upload files, navigate between folders, removing and renaming files, and open notebooks (.ipynb file extension).

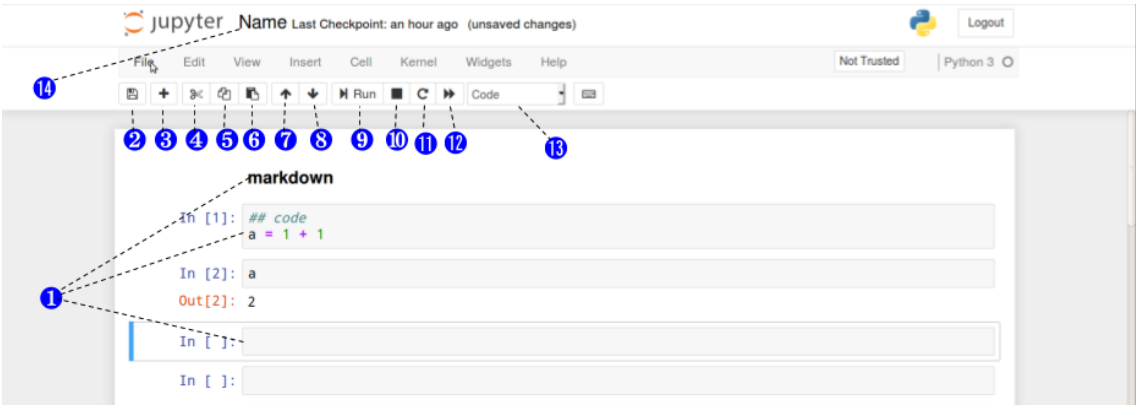
5. If you want to use one of the example datasets you can jump over this step. However if you want to analyse or running an experiment on your own dataset, you should navigate into the datasets folder and upload your files with the upload button located in the right top corner.

6. Next, you need to determine that if your prediction type is regression or classification. Based on your decision, you should choose and open the corresponding notebook file.

7. After you have opened the notebook, you are able to run the code, make your own experience and edit it like every other python code. One might take advantage of modification of hyperparameters, preprocessing or evaluation steps.

8. If you are not experienced in using Jupyter notebooks, this brief explanation might be useful. In the toolbar, there are the most common actions what you can do with your notebook. The following list will introduce the most-used operations as well as the most important parts of a notebook:

2.4.1 Interface of Jupyter Notebook



1. Cell: the place of code and descriptions, with (13) you can set the type of the selected cells
2. Save
3. Insert a cell below the selected cell
4. Cut selected cells
5. Copy selected cells
6. Paste selected cells below
7. Move selected cells up
8. Move selected cells down
9. Run selected cells
10. Interrupt the kernel (stop the program)
11. Restart the kernel (you will lose your local variables and objects)
12. Restart the kernel and run the whole notebook
13. The type of selected cells, the two most popular types are code and markdown
14. Name of the notebook

3 Developer documentation

3.1 Problem

In Data Science, we often want to predict numerical attribute of instances based on an annotated sample datasets. It is a well-understood area, to predict scores of instances having numerical attributes, however, predict scores of textual datasets is a highly challenging task. But this ability would be useful in many real-life problems such as scoring the huge number of essays or determining one's sentiment based on his or her feedback on a film or product. In this paper, I propose a method to solve a part of this problem.

3.1.1 General problem specification

Given a set of N training examples of the form $f(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)g$ such that x_i is a textual data, and y_i is its score, the aim is to find a function $m : X \; ! \; Y$ which can estimate the scores of the total population with the least error of some $g : Y \quad Y \; ! \; \mathbb{R}$ loss function or with the maximum gain of some $g : Y \quad Y \; ! \; \mathbb{R}$ gain function.

3.1.2 Regression problem

If the range of m is quantitative and ordinal or continuous, the problem of finding m is a regression problem. This is the case, for instance, if want to predict the scores of essays.

3.1.3 Classification problem

If the range of m is qualitative or nominal the problem of finding m is a classification problem. Like the majority of classification models, this model is also not able to handle more than 2 classes so the classification algorithm can only predict binary values. However as it is able to predict the probability of an instance belonging to class 1, its capabilities can be extended with the one-vs.-rest strategy, which trains a binary classifier for each different classes by converting their labels to class 1 and the rest of the classes to class 0. In this way, all binary classifier will predict the probability of an instance belonging to that certain class. So if we apply the ArgMax function on these binary classifiers we can make a prediction on classes of multiclass datasets as well.

3.1.4 Used Metrics

For **regression**, the two most popular loss functions, which we want to minimize over the population, are the following two

MAE (Mean Absolute Error)

MSE (Mean Squared Error)

RMSE (Root Mean Squared Error)

For **classification**, however we want to maximize the following scores over the population:

Accuracy

Precision

Recall

F1 score

3.2 Used methods and models

3.2.1 Evaluation metrics

For regression problems, we usually express the error of the model instead of the accuracy, which is hardly interpretable in this case. If the error is smaller, the model is better. However, we can use score maximization techniques if we change the sign of the metric to negative. One should thoroughly consider which loss function fit better on the given task. If one wants to penalty all deviation uniformly MAE can be a good choice, but if the larger deviation is worse than the small RMSE is better (if the range is not between 0 and 1).

Mean Absolute Error

$$\frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|$$

Mean Squared Error

$$\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Root Mean Squared Error

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}$$

For classification problems, one of the most natural and widely known metrics is accuracy, defined as the percentage of correct predictions. It is a sufficient basic metric, though it can be misleading when the number of instances in different classes are unbalanced. To illustrate this, if the task is finding the diseased people in a sample where only 1 per cent of the total is ill, then the model which predict all people to be healthy accuracy will be 99%, which can be very misleading if someone does not know the database. So it is very important to examine the database and check more metric. The combination of the following metrics can help one to measure the goodness of a certain model. Note that for these metrics, the higher the value, the better the performance.

Accuracy
$$\frac{\sum_{i=1}^{n}[y_i = \hat{y}_i]}{n}$$

Precision
$$\frac{\sum_{i=1}^{n}[y_i = \hat{y}_i = 1]}{\sum_{i=1}^{n}[\hat{y}_i = 1]}$$

Recall
$$\frac{\sum_{i=1}^{n}[y_i = \hat{y}_i = 1]}{\sum_{i=1}^{n}[y_i = 1]}$$

F1 score
$$2\,\frac{precision \cdot recall}{precision + recall}$$

Where y_i and $\hat{y}_i$ are the true and the predicted values, respectively.

3.2.2 N-grams

An n-gram is a contiguous sequence of n characters which appear in a given text. The n-gram representation of a given text is the set of all n-gram appearing in it. For extracting n-grams from a text, we usually iterate over the text with a window of size n and insert all appearing string to the set of representation. In this way, we have a set of strings instead of a raw text to represent a certain document. This representation has a hyperparameter n, which indicate the size of the window ergo the length of strings in the set as well. Naturally, n must be larger than 1, to be an integer and all text should be consist of at least n characters.

It is important to choose n properly as if we pick n too small, all documents will appear to be similar to each other, in contract, if we pick it too large, even similar documents will have a totally different representation. The aim is to choose n large enough to have the probability of an n-gram occurring in a text low but not too low. As a result, to choose an optimal window size, one should consider the cardinality of the character set and the average length of texts in your dataset.

For example, if someone pick n to be 1, probably all document will look like to be identical to each other as all document representation will be, more or less, a set of the characters from the alphabet. However, if we pick n to be too large, only the totally identical texts will have similar set representations and then we do not gain anything with this procedure.

I found n to be optimal when I picked it to be 2 for both the essay and the troll datasets.

3.2.3 Word2vec

Word2Vec is a relatively new cutting-edge technology which was published in 2013 by Tomáš Mikolov [1]. It is a model that is used to produce word embeddings, with other words, this model creates a function to map words to a high dimensional vector space in a way that words appearing frequently in similar contexts will be mapped close to each other. A context of a word is defined as a certain number of words before and after the word in a text from the given database. This number is usually between 5 and 10. The dimensionality of the vector space is usually between 100 and 1000, however, I will use a 300-dimensional model, pre-trained by Google because it is a computationally expensive process. Finally, as this model is not implemented by me and the performance of this representation is worse than the n-gram, I will regard this method as a black box.

3.2.4 MinHash

MinHash is an approximation technique for estimating the Jaccard index of two sets, where Jaccard index is a similarity measurement of sets, also called intersection over union [2]. Usually, the sets are represented with a fingerprint of size n, where n denotes the number of permutations. Generally, if n is larger, the performance is better but the running time is longer. To choose an optimal permutation number, one should strike a balance between running time and the probability of a good approximation.

$$J(X;Y) = \frac{jX \setminus Yj}{jX [Yj} = \frac{jX \setminus Yj}{jXj + jYj + jX \setminus Yj}$$

The algorithm A basic version of MinHash algorithm assigns a unique number to each element of the universal set from 0 to n-1 randomly, where n is the cardinality of the universal set. With other words, it picks a random permutation of the elements and assigns them their sequential number as their value. As random permutation is min-wise independent, each element of the universal set is equally likely to be assigned with the minimum value (0). The minhash value of a given set with a certain

permutation is the minimum value which has been assigned to one of the elements of the set by the random permutation process. The probability of two sets having the same minhash value is the Jaccard index of them. With more random permutations, precision can be boosted. Minhash values, produced by different permutations, of a set can be regarded as a fingerprint. Fingerprints produced this way are able to represent the sets to estimate their Jaccard indices well with each other. Computing random permutations on the universal set is usually infeasible and ineffective so instead of that, an improved version of MinHash uses nearly min-wise independent hashing to approach the theoretical performance of random permutations.

Set of n-grams When the elements of the sets are strings, we can take advantage of the wide range of cryptographic hash functions whose aim is to be uniformly distributed and easily computable. However the below-described algorithm is not min-wise independent, it still performs well according to others and in my experiments as well. Firstly, we use MD5 or SHA256 functions to map (hash) a string to an integer, so we can compute a numeric value for each string occurring in the sets. Next, we simulate random permutations by generating hash functions to map the values of strings to the set $f0;:::;n \quad 1g$ in different ways.

$$h(x) = (a \quad x + b) \bmod n$$

Where a and b are random integer parameters and n is the cardinality of the universal set.

Set of vectors When the elements of the sets are m-dimensional vectors and we do not want to treat these vectors as nominal values but we want to assign the same values to similar vectors, we can use the following method to produce hash functions mapping m-dimensional vectors to the set $f0;:::;n \quad 1g$.

$$h(x) = b(2 \quad \frac{x^\intercal r}{jxj \quad n} + \frac{n}{2})e \bmod n$$

Where r random m-dimensional unit vector and n is the length of the integer range.

3.2.5 LSH

Locality-sensitive hashing (LSH) is a dimensionality reduction technique whose purpose is recommending "candidate pairs" from a set, which pairs are likely to be similar to each other [2]. In the traditional way, if we want to find the most similar items in a set of size n, we need to do $\frac{n^2}{2}$ comparison, which is infeasible in case of large databases. In contrast to that, LSH is linear in the number of items, so with LSH, we are able to significantly reduce the number of comparisons. Generally, LSH

uses special hash functions, corresponding to the type of the objects in the set, to hash elements into buckets such a way that similar items are in the same bucket with higher probability than the dissimilar. If two elements are hashed into the same bucket at least once by one of the hash functions then they will be regarded as a candidate pair. The next step is not obligatory, however, it can significantly increase the precision of the model by discarding false positives from the candidate pairs by checking their real similarity score. So as, probably, there will be false-positives in the set of candidate pairs, to filter them out, we can check how similar they are in reality with a corresponding similarity function. If the similarity of a candidate pair is less than a certain threshold, which also can be a hyperparameter of the model, then that pair will be discarded.

As in my thesis, the model uses LSH for MinHash fingerprints, henceforth I will focus on LSH in terms of MinHash. Therefore the items are n-dimensional vectors, called fingerprints, where n is the permutation number of MinHash. To generate candidate pairs, LSH splits each fingerprint to band size slices and maps these slices into the identical bucket (implemented with dictionaries). So basically, two items will be a candidate pair if they have at least one identical slice. It is important to mention that, permutation number must be divisible by the band size.

It is crucial to choose band size properly as if it is too small, there will be a lot of matches and, naturally, more false positives, however, if it is too large, lots of similar items will not be suggested as a candidate. Thus, to find an optimal band size, one should strike a balance between too more and too few matches.

3.2.6 Semantic Graph

Semantic Graph is the fantasy name of our publicated method whose purpose is predicting the scores of previously unseen texts in a supervised manner. This model uses all the above-described methods to reach its goal, however, it is not just a combination of them. Firstly, it represents each text with a set of n-grams or a set of words. Secondly, it computes the fingerprint of each set with the corresponding version of MinHash. Thirdly, it builds an updateable and queryable LSH model with the training dataset and queries all candidate pairs having a larger Jaccard index than a certain threshold. Based on these pairs, it builds an attributed graph, where the nodes are the instances of the training dataset, and if there is an edge between two arbitrary nodes, then they are regarded to be similar. Edges of the graph can be weighted by the Jaccard indices of the fingerprints of the adjacent nodes. For now, we have a trained model. If we want to predict the score of an arbitrary text, the model will compute its MinHash fingerprint in the same way as it did with the training instances. Next, it queries LSH by the fingerprint to get a list of nodes

having fingerprints similar to the computed fingerprint. So basically, the model will do the same process with the test dataset except that it just queries the LSH model for nodes and does not modify that. So we have a list of nodes whose texts are likely to be similar to the given text. Optionally, we can expand this list with their first order neighbourhood so that the impact of scores of nodes which are reachable multiple times with one additional step will be more significant. Finally, the model predicts the score of the given text by the (un)weighted average of the scores of the nodes from the list. If it is a classification problem, this score is a prediction of the probability of the given text belonging to class 1. We can pick a threshold between 0 and 1 above which the text will be labelled with class 1. If the model can not find any similar node to a given text, also called unknown node, the score of the text will be predicted to either the mean of the scores of isolated nodes from training dataset or None, depending on the hyperparameter.

3.3 Project structure

3.3.1 Package diagram

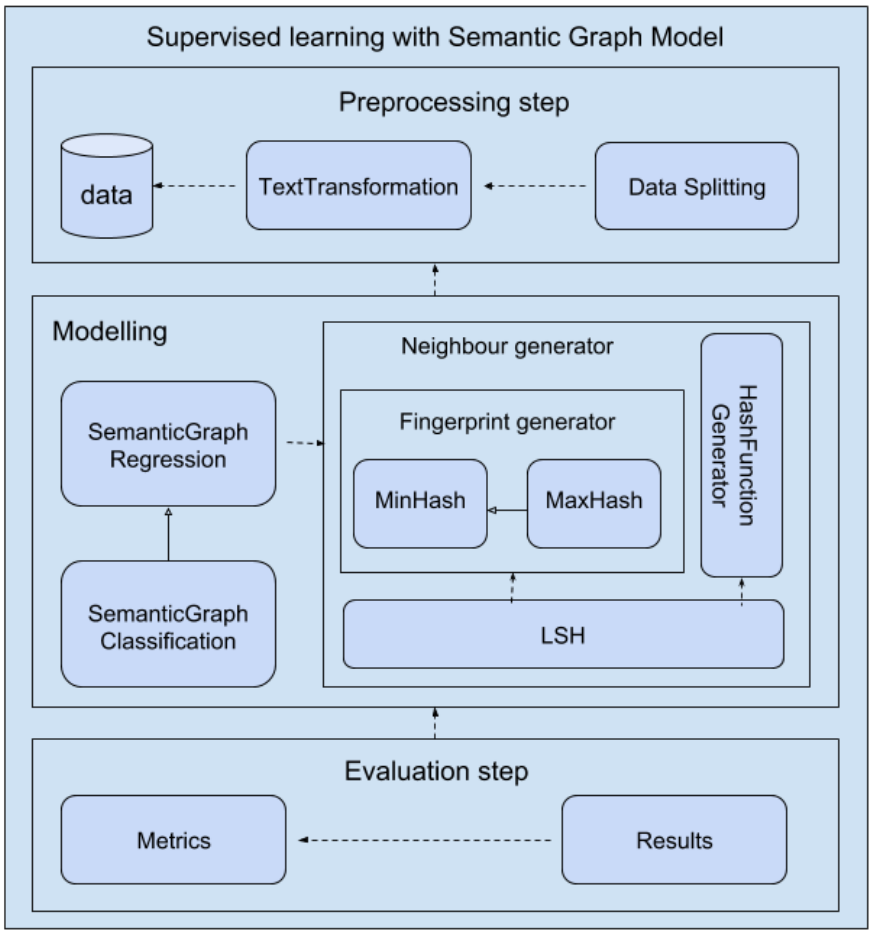
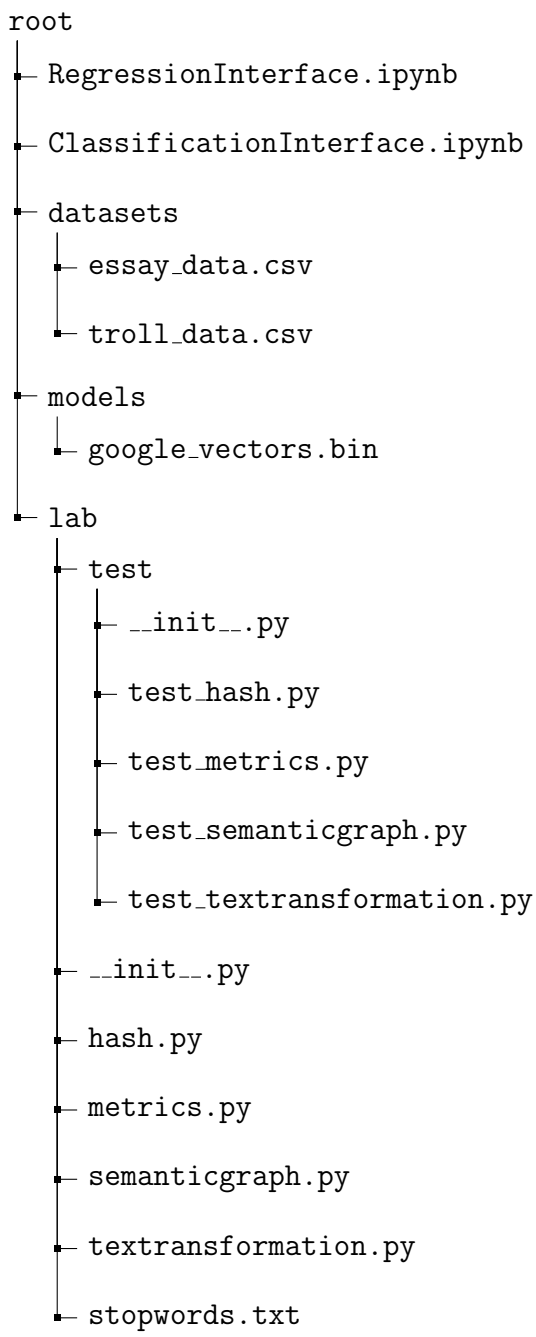


Figure 6: Package diagram

The following diagram describes the relationships of the implemented parts of the proposed method. As I previously mentioned, Data Science projects usually can be divided into three important steps. Having grouped the modules in this way, the first step is preprocessing which consists of transformation and splitting of the input dataset. TextTransformation is a class, and "Data Splitting" denotes a simple function which splits the transformed dataset into training and test datasets. The second step is modelling, including one class out of SemanticGraphRegression and SemanticGraphClassification classes depending on the type of the chosen task. These classes use LSH class to find the similar texts in order to build "semantic graph", and LSH uses MinHash or MaxHash classes to compute the fingerprints of texts by some corresponding hash functions which are generated by HashFunctionGenerator class. The final step is evaluation, therein some corresponding evaluation metrics show the performance of the trained model on the input dataset.

3.3.2 File tree

The following file tree illustrate the physical structure of the project. You can see from the diagram that, the input datasets are in folder "datasets", however the trained Word2Vec model and the list of stopwords are located in a separate folder "models". This can help us to distinguish the input datasets from the third-party datasets and models which are used by our model. In lab folder, the implemented classes and functions are. "hash.py" consists of HashFunctionGenerator, MinHash, MaxHash and LSH classes. "metrics.py"



3.3.3 Class diagram

SemanticGraphRegression and LSH classes have overlong parameter lists so some parameters are not shown in the diagram below. Instead of the missing parameters, args parameter is shown.

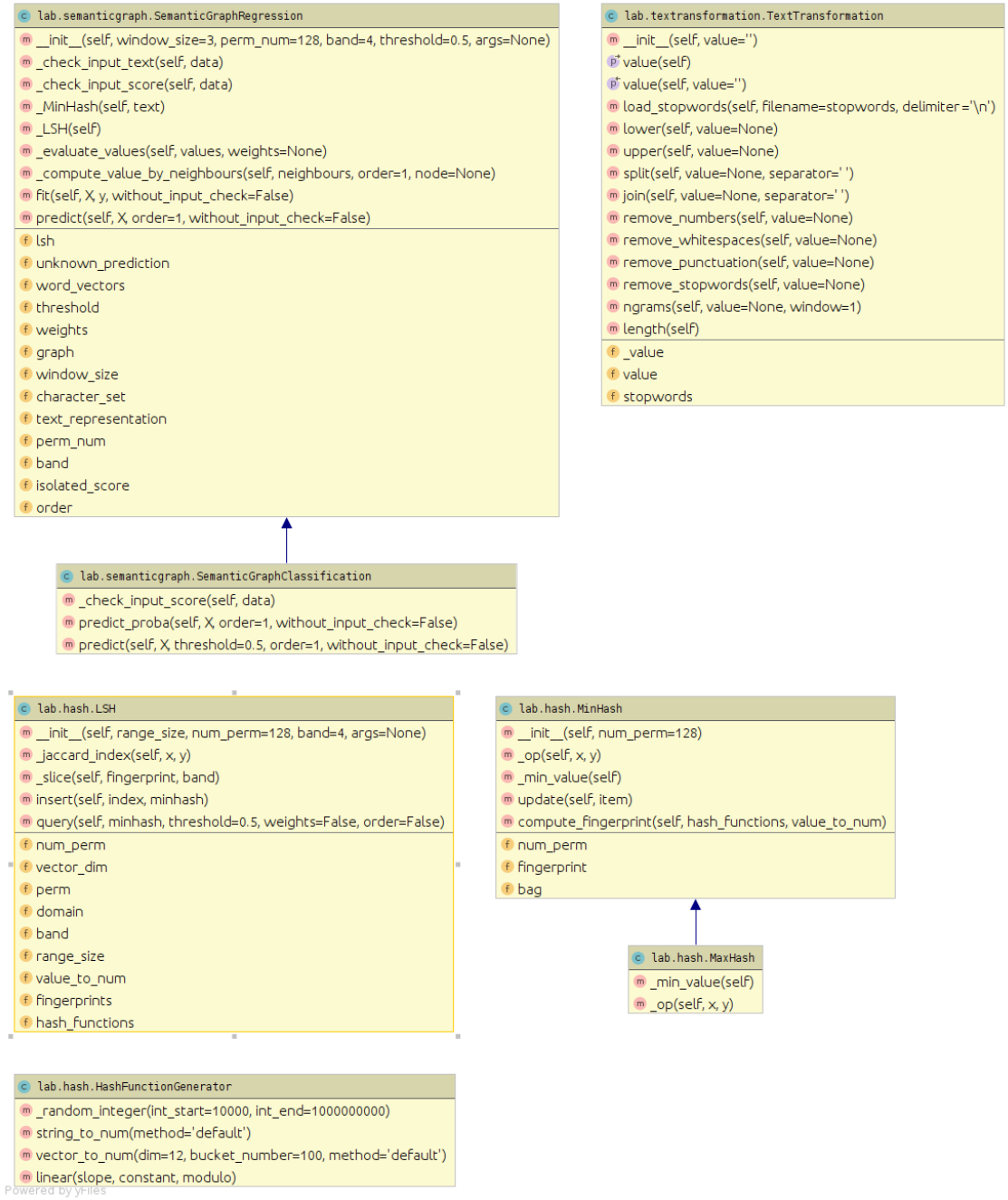


Figure 7: Class diagram

3.4 Classes

3.4.1 HashFunctionGenerator

The class purpose is providing an interface for different types of hash functions for MinHash, MaxHash and LSH classes.

Methods

```
string_to_num(method='default'):
```

String to number mapping function generator
Arguments

> *method* *f*'default', 'sha256', 'md5'*g* – chosen hash function ('default' = 'sha256')

Returns

> lambda – hashing strings to integers

```
vector_to_num(dim=12, bucket_number=100, method='default')
```

Vector to number mapping function generator
Arguments

> *dim* *f*int*g* – dimension of vector (default: 12)
>
> *bucket_number* *f*int*g* – the cardinality of the generated hash functions' codomain (default: 100)
>
> *method* *f*'default', 'dot'*g* – chosen hash function ('default' = 'dot')

Raises

> TypeError – *dim* must be an integer
>
> ValueError – *dim* must be 1
>
> TypeError – *bucket_number* must be an integer
>
> ValueError – *bucket_number* must be 1

Returns

> lambda – hashing vectors to numbers

```
linear(slope, constant, modulo)
```

A linear hashing function generator

Arguments

slope *f*int*g* – linear coefficient

constant *f*int*g* – constant term

modulo *f*int*g* – divisor

Raises

TypeError – *slope* must be an integer

TypeError – *constant* must be an integer

TypeError – *modulo* must be an integer

ValueError – *modulo* must be 1

Returns

lambda – hashing numbers into buckets in the interval

3.4.2 MinHash

The purpose of the class is representing a set with an integer vector, also called fingerprint, by random hash functions.

Class variables

num_perm – number of permutations

fingerprint – the computed fingerprint of the bag

bag – the set of items updated into the MinHash model

Methods

```
_init__(self, num_perm=128)
```

Arguments

num_perm *f*int*g* – number of permutations (or hash functions) (default: 128)

Raises

TypeError – *num_perm* must be an integer

ValueError – *num_perm* must be 1

ValueError – At least 1 element must be updated by the "update" method' (size of *bag* is 0)

```
update(self, item)
```

Update the bag (set) by adding an item to that
Arguments

*item f*hashable type*g* – a new item of the set

```
compute_fingerprint(self, hash_functions, value_to_num)
```

Compute fingerprint by the hash functions and the item *!* number function
Arguments

*hash_functions f*list*g* – list of different hash functions for generating fingerprint

*value_to_num f*lambda*g* – function for representing the items with numbers or vectors

Raises

TypeError – *hash_functions* must be a list

ValueError – the length of *hash_functions* must be equal with the length of *perm_num*

Returns

list – fingerprint (integer list of size *perm_num*)

3.4.3 MaxHash

MinHash class with a reversed internal operator, so the model chooses the maximal values of hash functions on the given set.

3.4.4 LSH

Class variables

range_size – the cardinality of the range of hash functions

num_perm – number of different permutations

band – length of buckets

domain – type of items (strings or words)

vector_dim – the dimension of word vectors if the domain is words

fingerprints – dictionary which stores the computed fingerprints by their id

hash_functions – generated hash functions for computing fingerprints

perm – bucket space for each hash function

Methods

```
__init__(self, range_size, num_perm=128, band=4,
   value_to_num='default', domain='strings', vector_dim=None)
```

Set up the environment for Locality-sensitive hashing and generate the corresponding hash functions

Arguments

range_size (int) – range size of hash functions

num_perm (int) – number of different permutations (or hash functions) (default: 128)

band (int) – length of buckets (default: 4)

value_to_num ('default', 'sha256', function) – function for representing the items with numbers or vectors ('default' = 'sha256')

domain ('words', 'strings') – type of items (default: 'strings')

vector_dim (NoneType, int) - dimension of word vectors if domain is words

Raises

TypeError – *range_size* must be an integer

ValueError – *range_size* must be 1

TypeError – *num_perm* must be an integer

ValueError – *num_perm* must be 1

TypeError – *band* must be an integer

ValueError – *band* must be 1

ValueError – *num_perm* must be divisible by *band*

TypeError – *vector_dim* must be an integer if domain is words

ValueError – *vector_dim* must be 1

```
insert(self, index, minhash)
```

Insert a MinHash instance, and map it into the corresponding buckets.

Arguments

*index f*hashable type*g* – unique key for identifying a minhash

*minhash f*MinHash*g* – minhash representation of a set

Raises

TypeError – *index* must be hashable

ValueError – *index* must be unique

TypeError – *minhash* must be an instance a class inherited from MinHash

TypeError – *minhash.num_perm* must be equal with *num_perm*

```
query(self, minhash, threshold=0.5, weights=False,
   order=False)
```

Returns the keys of sets having minhashes similar to the minhash argument at least with the passed threshold

Arguments

*minhash f*MinHash*g* – MinHash object for searching similar items in the LSH model

*threshold f*float : $[0; 1]$ *g* – filter out minhashes having less Jaccard indices with *minhash* (default: 0.5)

*weights f*bool*g* – should the returning list contains the Jaccard indices of MinHashes of the chosen keys? (default: False)

*order f*bool*g* – should the keys be ordered by their Jaccard index? (default: False)

Raises

TypeError – *minhash* must be an instance a class inherited from MinHash

TypeError – *minhash.num_perm* must be equal with *num_perm*

TypeError – the *threshold* must be a float

ValueError – the *threshold* must be from the range [0; 1]

TypeError – *weights* must be a bool

TypeError – *order* must be a bool

Returns

list – list of indices of similar items if *weights* == False

list – list of (index, similarity) pairs if *weights* == True

3.4.5 TextTransformation

Class variables

value – stores the current state of the transformation (it can be string or list of strings)

Methods

```
__init__(self, value='')
```

Set up the environment for transformation, and set an initial value for *value*

Arguments

value *f*str, list of str*g* – string or list of strings for transformation

```
load_stopwords(self, filename=stopwords, delimiter='\n')
```

Load stopwords for remove_stopwords method

Arguments

filename *f*str*g* – the filename of stopwords file (default: 'stopwords.txt')

delimiter *f*str*g* – delimiter of words in the file (default: '*n*n')

Raises

TypeError – *filename* must be an str

TypeError – *delimiter* must be an str

Returns

Self

```
lower(self, value=None)
```

Convert the text to lowercase

Arguments

value *f*str, NoneType*g* – the new text for transformations (default: None)

Raises

TypeError – self.*value* must be str for this method

Returns

Self

```
upper(self, value=None)
```

Convert the text to uppercase

Arguments

value *f*str, NoneType*g* – the new text for transformations (default: None)

Raises

TypeError – self.*value* must be str for this method

Returns

Self

```
remove_numbers(self, value=None)
```

Remove numbers from the text

Arguments

value *f*str, NoneType*g* – the new text for transformations (default: None)

Raises

TypeError – self.*value* must be str for this method

Returns

Self

```
remove_whitespaces(self, value=None)
```

Remove whitespaces from the text

Arguments

value *f*str, NoneType*g* – the new text for transformations (default: None)

Raises

TypeError – self.*value* must be str for this method

Returns

Self

```
remove_punctuation(self, value=None)
```

Remove punctuation from text

Arguments

value *f*str, NoneType*g* – the new text for transformations (default: None)

Raises

TypeError – self.*value* must be str for this method

Returns

Self

```
remove_stopwords(self, value=None)
```

Remove stopwords from text. If *self*.*stopwords* is None, it calls load_stopwords method with default parameters for loading the list of stopwords.

Arguments

value *f*str, NoneType*g* – the new text for transformations (default: None)

Raises

TypeError – self.*value* must be str for this method

Returns

Self

```
split(self, value=None, separator=' ')
```

Split the string into a list of strings by the separator

Arguments

value [str, NoneType] – the new text for transformations (default: None)

separator [str] – separator character or string (default: ' ')

Raises

TypeError – self.*value* must be an str for this method

Returns

Self

```
join(self, value=None, separator=' ')
```

Concatenate text pieces to a text

Arguments

value [str, list of str, NoneType] – the new list of strings for transformations (default: None)

separator [str] – separator character or string (default: ' ')

Raises

TypeError – self.*value* must be a list or an str for this method

Returns

Self

```
ngrams(self, value=None, window=1)
```

Produce a list of substrings of length 'window' which occurs in the text

Arguments

value [str, NoneType] – the new text for transformations (default: None)

window [int] – the length of substrings (default: 1)

Raises

TypeError – self.*value* must be str for this method

TypeError – *window* must be an int

ValueError – *window* must be ¿= 1

Returns

Self

```
length(self)
```

return the length of the value

Returns

int – length of text or list

3.4.6 SemanticGraphRegression

This class is an implementation of a supervised model for the score prediction of texts main steps: (1) Build a 'semantic' graph based on the similarities of texts. (fit method) (2) Score texts based on the semantic graph (predict method)

Class variables

text_representation – the form to which the texts will be converted (ngrams or words)

window_size – length of n-grams if *text_representation* is ngrams, otherwise the average length of the words

perm_num – length of minhash fingerprints

band – band parameter for LSH (length of buckets)

threshold – threshold parameter for LSH

weights – if it is True, the mean of the values of the neighbour nodes will be weighted by their Jaccard Index

order – the maximum distance of nodes, in the semantic graph, taking into consideration when the model predicts the value of a certain instance

character_set – the estimated size of the character set of texts

unknown_prediction – how to predict scores of nodes without neighbours

word_vectors – trained Gensim word2vec model if the *text_representation* is words

Methods

```
__init__(self, window_size=3, perm_num=128, band=4,  ·
   threshold=0.5, order=1, weights=False, character_set=26,  ·
   text_representation='ngrams', unknown_prediction='mean',  ·
   word_vectors=None)
```

Set the hyperparameters of the model and the trained word vectors if the *text_representation* is vectors

Arguments

window_size (int) – length of n-grams if *text_representation* is ngrams, otherwise the average length of the words (default: 3)

perm_num (int) – length of minhash fingerprints (default: 128)

band (int) – band parameter for LSH (length of buckets) (default: 4)

threshold (float) – threshold parameter for LSH (default: 0.5)

order (1, 2) – the maximum distance of nodes, in the semantic graph, taking into consideration when the model predicts the value of a certain instance (default: 1)

weights (bool) – if it is True, the mean of the values of the neighbour nodes will be weighted by their Jaccard Index (default: False)

text_representation ('ngrams', 'words') – the form to which the texts will be converted (default: 'ngrams')

character_set int – the estimated size of the character set

unknown_prediction 'mean', None – how to predict scores of nodes without neighbours (unknowns) (default: 'mean')

word_vectors (Word2VecKeyedVectors) – trained Gensim word2vec model (default: None)

Raises

TypeError – *perm_num* must be an int

ValueError – *perm_num* must be 1

TypeError – *band* must be an int

ValueError – *band* must be 1

ValueError – $perm_num$ must be divisible by *band*

TypeError – *threshold* must be a float

ValueError – *threshold* must be ¿= 0 and ¡= 1

TypeError – *order* must be an int

ValueError – *order* must be 1 or 2

TypeError – *window_size* must be an int

ValueError – *window_size* must be 1

TypeError – *character_set* must be a int

ValueError – *character_set* must be 1

ValueError – a trained word vector dictionary must be passed if the *text_representation* is vectors

```
fit(self, X, y, without_input_check=False)
```

Train the semantic graph model by example input-output pairs with the initialized hyperparameters

Arguments

X *f*list*g* – training data, list of texts (strings)

y *f*list*g* – target values (scores), list of floats or integers

without_input_check *f*bool*g* – should the training data be checked before the execution or not (default: False)

Raises

TypeError – *without_input_check* must be a bool

TypeError – *X* must be a list

TypeError – *y* must be a list

ValueError – lengths of *X* and *y* must be the same

ValueError – lengths of *X* and *y* must be 1

```
predict(self, X, order=1, without_input_check=False)
```

Predict the scores of texts from the given list

Arguments

X *f*list*g* – list of texts for prediction

order *f*1, 2*g* – how distant neighbours we want to take into account (default: 1)

without_input_check *f*bool*g* – should the training data be checked before the execution or not (default: False)

Raises

TypeError – *without_input_check* must be a bool

TypeError – *order* must be an int

TypeError – *X* must be a list

ValueError – *order* must be 1 or 2

Returns

list – list of predicted scores

3.4.7 SemanticGraphClassification

Binary classifier of texts based on SemanticGraphRegression class.

Methods

```
predict_proba(self, X, order=1, without_input_check=False)
```

Predict the probabilities of texts that they are belonging to class 1

Arguments

X *f*list*g* – list of texts for prediction

order *f*1, 2*g* – how distant neighbours we want to take into account (default: 1)

without_input_check *f*bool*g* – should the training data be checked before the execution or not (default: False)

Raises

TypeError – *without_input_check* must be a bool

TypeError – *order* must be an int

TypeError – *X* must be a list

ValueError – *order* must be 1 or 2

Returns

list – list of predicted probabilities

```
predict_proba(self, X, order=1, without_input_check=False)
```

Predict the classes of texts by calling *predict_proba* method with the corresponding parameters and a probability threshold.

Arguments

X *f*list*g* – list of texts for prediction

threshold *f*float*g* – the threshold for probability above which a class will be predicted to be 1.

order *f*1, 2*g* – how distant neighbours we want to take into account (default: 1)

without_input_check *f*bool*g* – should the training data be checked before the execution or not (default: False)

Raises

TypeError – *without_input_check* must be a bool

TypeError – *order* must be an int

TypeError – *X* must be a list

ValueError – *order* must be 1 or 2

TypeError – *threshold* must be a float

ValueError – *threshold* must be picked from the interval [0, 1]

Returns

list – list of predicted probabilities

3.5 Functions

3.5.1 metrics.py

This file provides an interface for the main classification and regression evaluation metrics

```
accuracy_score(true_values, predicted_values):
```

Compute the Accuracy score

Arguments

true_values *f*list*g* – correct target values

predicted_values *f*list*g* – predicted values

Raises

TypeError – *true_values* must be a list

TypeError – *predicted_values* must be a list

ValueError – length of *true_values* and *predicted_values* must be equal

ValueError – length of *true_values* and *predicted_values* must be 1

Returns

float – accuracy score

```
precision_score(true_values, predicted_values):
```

Compute the Precision score

Arguments

true_values *f*list*g* – correct target values

predicted_values *f*list*g* – predicted values

Raises

TypeError – *true_values* must be a list

TypeError – *predicted_values* must be a list

ValueError – length of *true_values* and *predicted_values* must be equal

ValueError – each item of *true_values* and *predicted_values* must be 0 or 1

ValueError – length of *true_values* and *predicted_values* must be 1

Returns

float – precision score

```
recall_score(true_values, predicted_values):
```

Compute the Recall score

Arguments

true_values *f*list*g* – correct target values

predicted_values *f*list*g* – predicted values

Raises

TypeError – *true_values* must be a list

TypeError – *predicted_values* must be a list

ValueError – length of *true_values* and *predicted_values* must be equal

ValueError – each item of *true_values* and *predicted_values* must be 0 or 1

ValueError – length of *true_values* and *predicted_values* must be 1

Returns

float – recall score

```
f1_score(true_values, predicted_values):
```

Compute the F1 score

Arguments

true_values *f*list*g* – correct target values

predicted_values *f*list*g* – predicted values

Raises

TypeError – *true_values* must be a list

TypeError – *predicted_values* must be a list

ValueError – length of *true_values* and *predicted_values* must be equal

ValueError – each item of *true_values* and *predicted_values* must be 0 or 1

ValueError – length of *true_values* and *predicted_values* must be 1

Returns

float – F1 score

```
classification_report(true_values, predicted_values):
```

Compute the most frequent classification metrics and the contingency table and visualize it

Arguments

true_values *f*list*g* – correct target values

predicted_values *f*list*g* – predicted values

Raises

TypeError – *true_values* must be a list

TypeError – *predicted_values* must be a list

ValueError – length of *true_values* and *predicted_values* must be equal

ValueError – each item of *true_values* and *predicted_values* must be 0 or 1

ValueError – length of *true_values* and *predicted_values* must be 1

Returns

tuple – Accuracy, Precision, Recall and F1 scores

```
mean_absolute_error(true_values, predicted_values):
```

Compute the Mean Absolute Error (MAE)

Arguments

true_values *f*list*g* – correct target values, list of numeric values

predicted_values *f*list*g* – predicted values, list of numeric values

Raises

TypeError – *true_values* must be a list

TypeError – *predicted_values* must be a list

ValueError – length of *true_values* and *predicted_values* must be equal

ValueError – length of *true_values* and *predicted_values* must be 1

Returns

float – Mean Absolute Error

```
neg_mean_absolute_error(true_values, predicted_values):
```

Compute the Negative Mean Absolute Error (MAE)

Arguments

true_values *f*list*g* – correct target values, list of numeric values

predicted_values *f*list*g* – predicted values, list of numeric values

Raises

TypeError – *true_values* must be a list

TypeError – *predicted_values* must be a list

ValueError – length of *true_values* and *predicted_values* must be equal

ValueError – length of *true_values* and *predicted_values* must be 1

Returns

float – Negative Mean Absolute Error

```
mean_squared_error(true_values, predicted_values):
```

Compute the Mean Squared Error (MSE)

Arguments

true_values *f*list*g* – correct target values, list of numeric values

predicted_values *f*list*g* – predicted values, list of numeric values

Raises

TypeError – *true_values* must be a list

TypeError – *predicted_values* must be a list

ValueError – length of *true_values* and *predicted_values* must be equal

ValueError – length of *true_values* and *predicted_values* must be 1

Returns

float – Mean Squared Error

```
neg_mean_squared_error(true_values, predicted_values):
```

Compute the Negative Mean Squared Error

Arguments

true_values *f*list*g* – correct target values, list of numeric values

predicted_values *f*list*g* – predicted values, list of numeric values

Raises

TypeError – *true_values* must be a list

TypeError – *predicted_values* must be a list

ValueError – length of *true_values* and *predicted_values* must be equal

ValueError – length of *true_values* and *predicted_values* must be 1

Returns

float – Negative Mean Squared Error

```
root_mean_squared_error(true_values, predicted_values):
```

Compute the Root Mean Squared Error (RMSE)

Arguments

true_values *f*list*g* – correct target values, list of numeric values

predicted_values *f*list*g* – predicted values, list of numeric values

Raises

TypeError – *true_values* must be a list

TypeError – *predicted_values* must be a list

ValueError – length of *true_values* and *predicted_values* must be equal

ValueError – length of *true_values* and *predicted_values* must be 1

Returns

float – Root Mean Squared Error

```
neg_root_mean_squared_error(true_values, predicted_values):
```

Compute the Negative Root Mean Squared Error

Arguments

true_values *f*list*g* – correct target values, list of numeric values

predicted_values *f*list*g* – predicted values, list of numeric values

Raises

TypeError – *true_values* must be a list

TypeError – *predicted_values* must be a list

ValueError – length of *true_values* and *predicted_values* must be equal

ValueError – length of *true_values* and *predicted_values* must be 1

Returns

float – Negative Root Mean Squared Error

```
cross_validation_score(model, X, y,
   score_metric=accuracy_score, k=10):
```

Compute the k-fold cross validation score with the given metric

Arguments

model callable – The estimator object to use to fit the data

X list – the data to fit

y list – the target values

score_metric function – metric to evaluate the model

k int – the number of groups that the dataset is to be split into

Raises

TypeError – *X* must be a list

TypeError – *y* must be a list

ValueError – length of *X* and *y* must be equal

ValueError – length of *X* and *y* must be 1

TypeError – k must be an int

ValueError – k must be > 1

Returns

float – k-fold cross validation score

3.6 Testing

3.6.1 Test plan

Software testing is a very essential and important part of software development as the reliability of programs depends on it, moreover, it helps the programmers to identifying and fixing bugs in the code before they hand that out. However, it can be really monotonous to test code if it is done manually. Fortunately, one can make it automatic by writing scripts to check the correct operation of the independent parts of the program. These scripts are also called "unit tests". The future developments are also became easier as we can check the effect of an update on the whole code base immediately with automated tests.

In Data Science, testing can be even better challenging as machine learning models are usually stochastic processes, and they often suffer from the lack of mathematical proofs, plus the results are rather empirical than theoretical. This model is not different, and therefore, the aim is to do unit tests wherever it is possible to check anything from the list below:

the invariant

the returned value

the proper exception handling when the input is incorrect.

3.6.2 Unit tests

There are unit tests, organized into the "lab*n*test" folder, for each source code from the lab directory. They are implemented with the support of a popular unit testing framework "unittest", which is a standard module in Python. For each method and function, there are tests for checking the proper exception handling when the input is incorrect, and with the exception of *SemanticGraphRegression* and *SemanticGraphClassi cation* classes the most important invariant properties and returned values are also checked. Unfortunately, in the case of the above-mentioned classes, there is no well defined invariant properties to check.

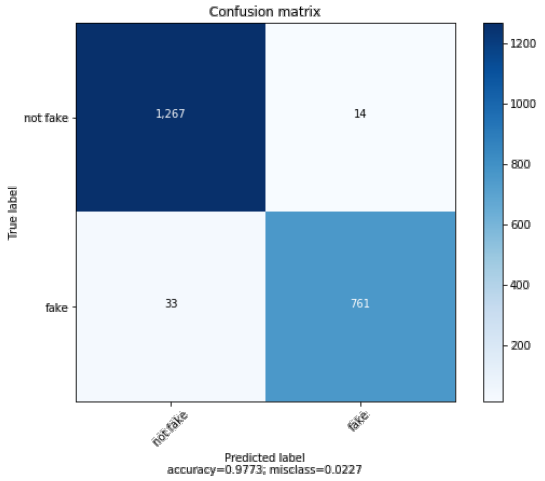
One can run all unit tests with the following command from the test folder:

```
python -m unittest
```

3.6.3 Experimental Results

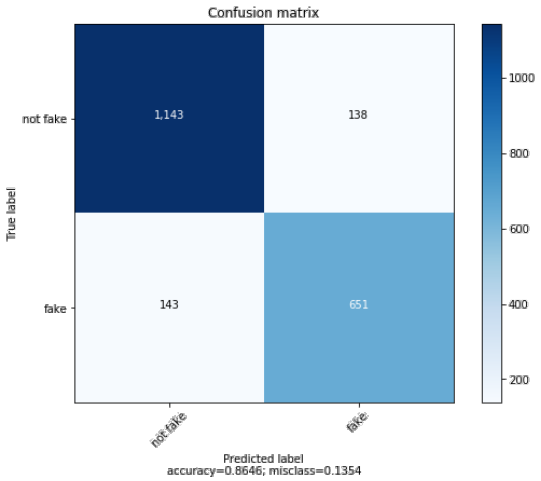
Though the main model (Semantic Graph) can not be tested in the traditional way, its empirical performance can be expressed by suitable evaluation metrics, in data science manner.

Classification The experiment was carried out on the dataset "Tweets Dataset for Detection of Cyber-Trolls" provided by DataTurks at Kaggle[3]. The dataset was split into a training set and a test set in 90:10 proportion.



	precision	recall	f1-score	support
0	0.97	0.99	0.98	1281
1	0.98	0.96	0.97	794
micro avg	0.98	0.98	0.98	2075
macro avg	0.98	0.97	0.98	2075
weighted avg	0.98	0.98	0.98	2075

(a) N-gram representation



	precision	recall	f1-score	support
0	0.89	0.89	0.89	1281
1	0.83	0.82	0.82	794
micro avg	0.86	0.86	0.86	2075
macro avg	0.86	0.86	0.86	2075
weighted avg	0.86	0.86	0.86	2075

(b) Word2Vec representation

Regression The experiment was carried out on the essay dataset provided by the Hewlett Foundation at Kaggle[4]. The dataset was split into a training set and a test set in 90:10 proportion.

text repr.	MAE	MSE	RMSE
N-gram	0.32579	0.14229	0.37722
Word2Vec	0.38691	0.22058	0.46966

[3]https://www.kaggle.com/dataturks/dataset-for-detection-of-cybertrolls
[4]https://www.kaggle.com/c/asap-sas

4 References

[1] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[2] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Finding Similar Items*, page 68–122. Cambridge University Press, 2 edition, 2014.