Course Project: Automated Fact Checking

Julien Nahum - SN: 15063949
zcabjna@ucl.ac.uk

For each of the questions, many intermediate files were generated to cache and speed-up recurrent or later computations.

Different code utilities were developed for this project. They are all located in the *lib* folder of the project. It contains the following files:

- : *datasets_parsing.py* that contains all functions used to load, cache and manipulate the different datasets of the project. It contains functions directly taken or modified from *fever_io.py*.
- : *graph.py* that contains all methods used to generate matplotlib charts.
- : *logistic_regression.py* that contains the Logistic regression class created for Q4.
- : *stats.py* that contains all mathematical methods created for Q2 and Q3 such as TF-IDF, cosine similarity and the different smoothing techniques.
- : *utils.py* that contains all utilities methods to manipulate strings, split words, or cache computations.

The main code for Q1 can be found in "q1.py". The main code for Q2 can be found in "q2.py". The main code for Q3 can be found in "q3.py". The main code for Q4 and Q5 can be found in "q4-q5.py". The main code for Q6 and Q8 can be found in "q3.py". All the required files (the two csv and the jsonl file) are in the results directory.

1 TEXT STATISTICS

A dictionary mapping words to number of occurrences in corpus was created. A total of 4.260.970 different words were identified. They were sorted by descending order of number of occurrences in the corpus. The figure 1 below shows the distribution of word frequencies. Because of the scale, the curve of the blue line on the bottom left part of the graph cannot be properly seen, but the distribution seem to respect Zipf's law.

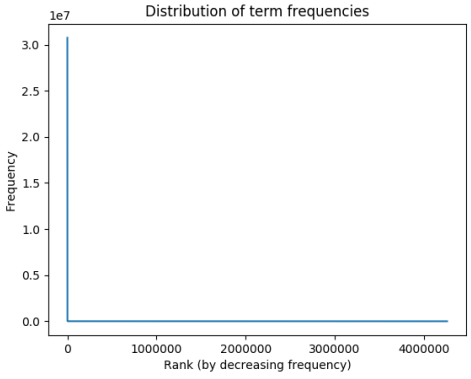

Figure 1: Distribution of word frequencies in Corpus

The figure 2 plots for each different word in the corpus, its rank multiplied with it's probability of occurrence. As it can be estimated from the graph, the average value of

$$c = rank * Pr$$

for this corpus is c=0.009041. This result is close to the value of 0.1 that would have an English corpus perfectly respecting Zipf's law. Additionnaly, the average

$$k = rank * frequency$$

was computed. It was found that the average k is 3.971.364 which is pretty close to 4.260.970, the number of different words in the corpus. In definitive, it seems that the Zipf's law is mostly verified in this corpus.

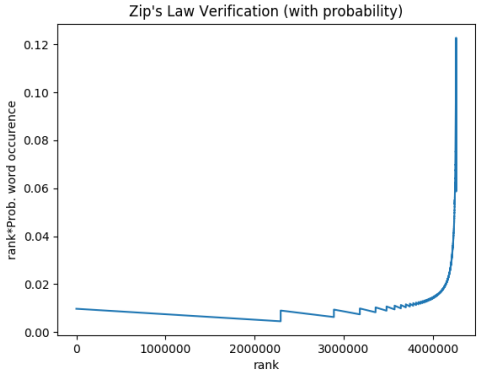

Figure 2: Verification of Zipf's law

2 VECTOR SPACE DOCUMENT RETRIEVAL

For this question, we have to extract TF-IDF representations of the 10 given claims and of all documents. Computing Term-Frequency is pretty straightforward, as it only required a document/claim:

$$TF(word) = \frac{\text{Number of times term word appears in a document}}{\text{Total number of words in the document}}$$

However, to compute Inverse-Document-Frequency, all documents of corpus must be accessed to check how often words are used.

$$IDF(word) = log(\frac{\text{Total number of documents}}{\text{Number of documents with word in it}})$$

In order to be more efficient I created a inverted index. It is a dictionary mapping all the different words in the corpus to a list of their occurrence through the different documents. It allows very fast access to the number of occurrence of a specific word in a specific document. It structure is as follows:

```
invertedIndex = {
  'word1' : [ (doc_id_1,4),(doc_id_2,4)],
  'word2' : [ (doc_id_3,1),(doc_id_2,2)]
}
```

where the second element of each tuple is the number of occurrence of the word in the related document. This inverted index allowed me to easily compute the docIdf, or simply the IDF for each word in the corpus.

Computing the TF-IDF of a document/claim was then very simple: for a given word simply multiply its TF and its IDF. Even though computing TF-IDF representation with the inverted index was really fast for 10 claims, it was not the case for the whole collection of documents. Therefore, in order to reduce useless computations I computed a set of "important documents". To do so, I created a set of all important words in the 10 claims, with a TF-IDF value greater than 0.2. I then went through all documents in the corpus, to filter out all the documents that did not contain at least one of the "important words". As a result, 728.446 documents were found, reducing by approximately 8 the number of later computations. For each of these documents the TF-IDF representations were computed.

Once that was done, calculating the cosine similarity between each document and each claim was fairly simple. For each claim, documents were ranked by cosine similarity to find the top 5 documents. The final results can be found in *q2.csv*.

3 PROBABILISTIC DOCUMENT RETRIEVAL

For this question I had to build a query-likelihood unigram language model. I decided to remove stop-words in order to optimize results. I order to restrict computations and to improve my results, I remove stop-words, and did computations only for the words presents in the claims. For each Wikipedia articles, I created 4 models (with the 3 different types of smoothing, and without any smoothing).

For the Laplace smoothing, I used the vocabulary size (4.260.970) that was already computed before. However, for the Jelinek-Mercer smoothing and the Dirichlet smoothing, I had to compute more metrics on the dataset such as the collection frequency (439.216.426) and the average number of words per document.For the Jelinek-Mercer smoothing, I used 0.5 as a constant, while for the Dirichlet smoothing I used the average number of words per document which I calculated to be 3572.

Once I had computed the 4 models (with and without smoothing) for each of the articles, I had to compute the query score of each claim in order to find the 5 most similar article for each of the 10 claims.

Result can be found in *q3.csv*.

4 SENTENCE RELEVANCE

I started this question by computing the top five relevant documents for 10.000 claims of the training subset, using a query likelihood model with Dirichlet smoothing. To avoid having to go through each documents for each claim, I pre-filtered the collection of documents, finding for each claims all documents that contained at least 2 different non-stop-words in common with the claim. I then built and used a query likelihood model with Dirichlet smoothing to find the five most relevant documents of each 10.000 claims. I then identified the claims for which all the evidences were taken from the top five documents I computed, and discarded the others. With the remaining claims, I built a balanced dataset by proceeding as follows: first collect all evidences as positive examples (relevant sentence), and then for each positive example, pick a random sentence from the previously computed top five documents as a negative example. Of course, no sentences were used twice.

Learning Rate	Epochs	Accuracy	Precision	Recall	F1 Score
0.005	1	0.576923	0.541667	1.000000	0.702703
0.005	10	0.576923	0.541667	1.000000	0.702703
0.005	100	0.576923	0.541667	1.000000	0.702703
0.005	1000	0.596154	0.580645	0.692308	0.631579
0.005	10000	0.596154	0.608696	0.538462	0.571429
0.01	1	0.576923	0.541667	1.000000	0.702703
0.01	10	0.576923	0.541667	1.000000	0.702703
0.01	100	0.519231	0.511111	0.884615	0.647887
0.01	1000	0.557692	0.555556	0.576923	0.566038
0.01	10000	0.596154	0.608696	0.538462	0.571429
0.05	1	0.576923	0.541667	1.000000	0.702703
0.05	10	0.576923	0.541667	1.000000	0.702703
0.05	100	0.596154	0.580645	0.692308	0.631579
0.05	1000	0.596154	0.608696	0.538462	0.571429
0.05	10000	0.634615	0.640000	0.615385	0.627451
0.1	1	0.576923	0.541667	1.000000	0.702703
0.1	10	0.519231	0.511111	0.884615	0.647887
0.1	100	0.557692	0.555556	0.576923	0.566038
0.1	1000	0.596154	0.608696	0.538462	0.571429
0.1	10000	0.730769	0.700000	0.807692	0.750000
0.15	1	0.576923	0.541667	1.000000	0.702703
0.15	10	0.500000	0.500000	0.807692	0.617647
0.15	100	0.576923	0.576923	0.576923	0.576923
0.15	1000	0.615385	0.636364	0.538462	0.583333
0.15	10000	0.692308	0.656250	0.807692	0.724138
0.2	1	0.576923	0.541667	1.000000	0.702703
0.2	10	0.500000	0.500000	0.807692	0.617647
0.2	100	0.576923	0.583333	0.538462	0.560000
0.2	1000	0.615385	0.625000	0.576923	0.600000
0.2	10000	0.673077	0.636364	0.807692	0.711864

Table 1: Performance of Logistic regression with the balanced dataset

To better study performance of the logistic regression I implemented, I created two test datasets: one balanced and one unbalanced. The balanced dataset was created by taking all 10 claims' evidences as positive elements of the dataset, and by randomly taking as many sentences from the evidence documents as negative sample. To created the unbalanced dataset, I did the same thing, but took all remaining sentences as negative examples. For each elements of dataset (training included), I removed the claim and the sentence's stop words, and concatenated the two list of words obtained.

Learning Rate	Epochs	Accuracy	Precision	Recall	F1 Score
0.005	1	0.400000	0.273684	1.000000	0.429752
0.005	10	0.400000	0.273684	1.000000	0.429752
0.005	100	0.417391	0.279570	1.000000	0.436975
0.005	1000	0.608696	0.327273	0.692308	0.444444
0.005	10000	0.686957	0.368421	0.538462	0.437500
0.01	1	0.400000	0.273684	1.000000	0.429752
0.01	10	0.400000	0.273684	1.000000	0.429752
0.01	100	0.400000	0.258427	0.884615	0.400000
0.01	1000	0.634783	0.326087	0.576923	0.416667
0.01	10000	0.686957	0.368421	0.538462	0.437500
0.05	1	0.400000	0.273684	1.000000	0.429752
0.05	10	0.417391	0.279570	1.000000	0.436975
0.05	100	0.608696	0.327273	0.692308	0.444444
0.05	1000	0.686957	0.368421	0.538462	0.437500
0.05	10000	0.713043	0.410256	0.615385	0.492308
0.1	1	0.400000	0.273684	1.000000	0.429752
0.1	10	0.400000	0.258427	0.884615	0.400000
0.1	100	0.634783	0.326087	0.576923	0.416667
0.1	1000	0.686957	0.368421	0.538462	0.437500
0.1	10000	0.721739	0.437500	0.807692	0.567568
0.15	1	0.400000	0.273684	1.000000	0.429752
0.15	10	0.408696	0.250000	0.807692	0.381818
0.15	100	0.669565	0.357143	0.576923	0.441176
0.15	1000	0.686957	0.368421	0.538462	0.437500
0.15	10000	0.704348	0.420000	0.807692	0.552632
0.2	1	0.400000	0.273684	1.000000	0.429752
0.2	10	0.452174	0.265823	0.807692	0.400000
0.2	100	0.660870	0.341463	0.538462	0.417910
0.2	1000	0.686957	0.375000	0.576923	0.454545
0.2	10000	0.704348	0.420000	0.807692	0.552632

Table 2: Performance of Logistic regression with the unbalanced dataset

The next step was to produce word embeddings for the training dataset and the testing datasets based using Word2Vec. To optimize the results, I decided not to train my own Word2Vec model. I chose to use Google's trained Word2Vec [8] which includes word vectors for a vocabulary of 3 million words and phrases that was trained on roughly 100 billion words from a Google News dataset. The vector length is 300 features. With the help of the gensim package, I did the embedding for each element of all 3 datasets.

I wrote a python class for the logistic regression. It accepts a learning rate and a number of feature as parameters of its constructor. When training, it also possible to specify the number of epochs. I tried the following learning rates: 0.005,0.01,0.05,0.1,0.15 and 0.2 along with the following number of epochs: 1,10,100,1000,10000. I studied the evolution of the model training loss with different learning rates. It seems that they whatever the learning rate is, the training loss decrease and converge to around 0.65. The only difference is the speed of convergence: the greater the learning rate is, the faster the training loss converges.

I evaluated the performance of the model using the 4 metrics in the question 5. Results can be found in the next question.

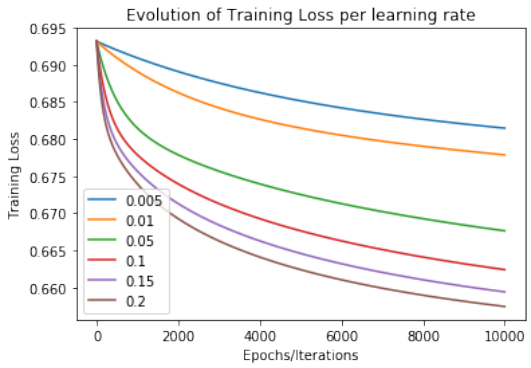

Figure 3: Evolution of Training Loss per Learning Rate

5 RELEVANCE EVALUATION

In this question, I implemented methods to compute recall, precision and F1 metrics. To do so, for each of the test dataset I counted the following: true positive - TP (relevant sentences predicted as relevant), false positive - FP (non-relevant sentences predicted as relevant), true negative - TN (non-relevant sentences predicted as non relevant) and false negative FN (relevant sentences predicted as non relevant).

Then, I implemented methods to computes the metrics using these formulas:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1 - Score = \frac{2 * (Recall * Precision)}{Recall + Precision}$$

It is interesting to note that for the two datasets, the model performed best with a learning rate of 0.1 and 10.000 epochs. Accuracy, Precision and F1 Score are generally lower for the unbalanced dataset. Accuracy and Recall are pretty similar for both datasets using the optimal settings found, but Precision and F1 Score are much better for the balanced dataset. That is not surprising because as we trained with an balance dataset, the number of false positive is greater for the unbalanced dataset. Therefore the precision and smaller, and so is the F1 Score.

Accuracy	0.6247
Precision	0.5748
Recall	0.9589
F1	0.7187

Table 3: Performance of final model

6 TRUTHFULNESS OF CLAIMS

For this fact cheking, I decided to use a Recurrent Neural Networks (RNN) and Long Short Term Memory (LSTM). Unlike other RNN algorithms, LSTM have the possibility to forget or update already learned information. For the implementation, I used Keras, a python deep-learning library that uses Tensorflow API. The main layer of the nework is the LSTM layer, but I also used an embedding layer to transform the network input into a format suitable for LSTM, a dense layer to change so dimension of the output vector, a dropoout layer to randomly set some dimensions of vectors to zero (in order to avoid overfitting) and finally an activation layer (a classic sigmoid to obtain a value between 0 and 1. However, my model does not predict evidence use for the decision.

I retrieved all the 87308 training claims (discarding the 'NOT ENOUGH INFOâĂŹ claims), and all the 13229 testing claims, and did the following for each: find the relevant sentences, and concatenate them to the claims. Then for the embedding of these claims, I first tried used the google trained word2vec model as I did in the previous questions, but results were not great. I changed the embedding to use the Tokenizer class that is in the Keras library. More details on this choice in question 8.

I then trained the model using a batch size of 128 and 20 epochs. The training dataset used had 79551 claims with label 'SUPPORTS' and 29584 claims with label 'REFUTES'. The testing dataset is balanced with 6615 label 'SUPPORTS' and 6614 label 'REFUTES'.

I used the metrics function implemented in question 5 to measure the predictions done by my model and obtained the following performances:

7 LITERATURE REVIEW

in 2016 an American father attacked a pizza shop located hundred of miles away from his home, because he read online that the restaurant was harboring young children as sex slaves [5]. Even though that kind of event is pretty unique, fake news misinformation have impact on everyone's daily life. Fact-checking is a task commonly performed by journalists or by users of web platform such as Politifact (US), or FactCheck.org. Fake news are commonly used by politicians and represent a really challenge for journalists. The main issue is that fact-checking is a very long task that cannot be done in real-time and therefore isn't really helpful most of the time. To help journalists, researchers of [14] created Claimbuster, a tool to help journalists identifying important claims in real-time. Interest for automated fact checking has been growing a lot in the last decade. Three main categories of fact-checking methods were identified: knowledge based (relating information to known facts), context-based (analyzing news spread in social media), and style-based (analyzing writing style). The paper [12] is a review of fake news detection methods, including data mining algorithms. It presents the two kind of methods used for feature extraction: linguistic based and visual based (for videos and images). It also presents methodology to detect fake profiles on social medias that bots use to spread fake news. It also details the two main task of Knowledge-based fact checking: identifying important claims, and checking their veracity with the help of knowledge graphs. Finally, it introduces the style-based methods and introduce commonly used evaluation metrics for fact-checking algorithms.

The paper [2] is a survey of methods created to recognize textual and non-textual clickbaits. Clickbaits content design all the content which are created is the main purpose to have a website visitor to click on it. These clickbaits content often present misleading and unverified information, that are used extensively in the propagation of fake news. Four main categories of methods are presented: lexical and semantic analysis against syntactic and pragmatic for the textual part, and image analysis and user behaviour for the non-textual methods. Only a description is provided for each of the 4 methods. No implementations or actual classification models are presented.

The papers [9] and [1] focus mainly on the style-based methodology. Researchers of [9] used various styling metrics to build a classification to predict partisanship (left or right) of a text author. Using this classification, they also tried to build a classifier to predict veracity of a claim. Their work shows that writing style in news articles is enough to distinguish a hyperpartisan world view from more balanced news. It was found that extremes are very similar and different from mainstream. In [1], a method is presented a method for detecting stylistic deception in written documents. With this technique, deceptive documents can be detected with 96.6% accuracy. 3 main categories of features are used by the three classification models built: Lexical features, Syntactic features and Content Specific features. This research does not really focus on truthfullness checking, but more on regular authorship recognition. The method presented in [10] also uses language and more precisely linguistic characteristics to build a classification of texts. It uses news identified as satire, hoaxes, and propaganda to find linguistic characteristics of untrustworthy text. Two major contributions are presented: a fake news analysis that uses a Max-Entropy classifier to distinguish and a truthfulness predictor using an LSTM model. Other model were tested for the truth-fullness detection, but LSTM outperformed them. Their result shows that various lexical features can contribute to the challenging fact-checking task.

Other papers were found focusing more intensively on knowledge based methods. In [4], two methods of veracity assessment are presented: linguistic cue approaches (ML) and network analyses approaches. The linguistic approach discuss different manners to represent data (such as bag of words). Researchers trained classifiers (SVM and NaÃŕve Bayesian models) to perform veracity assessment tasks. This approach is promising when combined with other approaches, but can hardly be generalized for real-time veracity detection of news. The network analysis approach links data and knowledge networks for the classification. The model built is pretty efficient (between 61% and 95% depending on model and parameters), but has an important limitation: facts must be pre-existing

knowledge base. Similarly, the paper [15] presents a method using K-nn trained with facts already checked. The main shortcoming, is that because it uses semantic similarity, it can only work with paraphrase version of already checked claims and not with new claims.

In [18] the researchers introduce a method to check claims veracity by perturbing queries performed to check the claims. This technique is also limited to by the fact that queries are checked against finite a database.//

The researchers of [3] used data from wikipedia to build a knowledge graph of 3 millions node and approximately 23 millions edges. They then used the built graph as follows: the more a node has connection in that graph, the more general it is and the weaker is the support it provides. It was found by researchers, that statements known to be true receive higher support than false ones.A shortcoming is that the built graph is not directed, and that therefore some information is lost.

The papers [17] and [13] both use social medias structure to detect fake news and hoaxes post. In [17], researchers analyses the propagation of misinformation using graph theory. They labelled users as susceptible, infections and recovered, and then studied how fake news spread. This method is not content-oriented at all (claims or articles are not analyzed) but focuses on relaying patterns and identifying weak nodes. A different approach was taken in [13]. They used Facebook likes to detect hoaxes post on Facebook with a very high accuracy (99%). Two model were tested: logistic regression and boolean crowd-sourcing algorithms. The set of users that interacts with news posts in social network sites are used to predict whether posts are hoaxes or not. The main limitation is that the users (1 milion in the dataset researcher used) are the characteristics of the regression and therefore could prevent this method to scale. However, the researchers affirm that with a really small dataset, the boolean crowdsourcing algorithms perform almost as well, and that it would be possible to scale that to the size of a social network such as Facebook. This paper highlights how the study of networks can reveal information regarding fact-checking.

Finally, many papers discussed and introduced neural network based methods to perform fact-checking. A problem encountered when trying to developp such models, is the lack of labeled training data. Researchers of [16] present their own dataset for fake news detection along with a hybrid convolutional neural network (CNN). They tested and compared different models (SVM, LR, Bi-LSTMs and various hybrid of CNNs) for fact-chcking and also used also using google news trained word2vec. They found that Bi-LSTMS tends to lead to overfitting and that CNNs outperfom all models. In [11], the researchers trained an SVM-based algorithm to detect an recognize satirical news. They used 5 predictive features (Absurdity, Humor, Grammar, Negative Affect, and Punctuation) to produce a final model that detects satirical news with a precision of 90%. The dataset used for testing and training only consists of 360 news article. Each satirical news was matched with a real news posted in the same country in the same period.

Researchers of [7] created a Bidirectional LSTM network that performs fact-checking prediction. The claims are transformed into various representations of words and characters to identify linguistic features common in false claims. They also used the Google trained word2vec in their embedding method. Three class are used in the classification (TRUE, HALF-TRUE and FALSE) and accuracy obtained is below 40%. That could be explained according to researchers by the lack of use of knowledge base, that could improve the performances of the model. In [6], the framework introduced also uses a deep neural network with LSTM and SVM to accomplish fact-checking of claims and rumor detection. Very interestingly, their model automatically uses different online search engines to find documents relevant documents helping to prove or disprove a claim. The entire Web is used as their knowledge source. The most relevant sentences from the web pages obtained are extracted and compared to the claim (using cosine and tf-idf). The best classifier found uses a mix of SVM and a neural network and has 72% of accuracy. The method is very lightweight as it only uses the page snippets presented by the search engine, yet it gives good results. This method could be a solution to the shortcomings raised in [4] and in [15].

The holy grale of fact-checking could be a hybrid techniques using knowledge based methods, context-based methods and style-based methods. Exploring automated knowledge source such as in [6] could really help improving results. Unsupervised learning could also be very interesting domain to explore for automated fact checking tasks.

8 IMPROVEMENTS

The first part that could be improved is the logistic regression. No changes were implemented for that, but there is definitely room for improvements. Using sentences2vec for instance could give better results than an averaged word2vec. Feature scaling or normalization could also be used to improve the current results. Other models such as classifications trees could also be tested for the sentence relevance part of this coursework.

For the truthfulness of claims part, few improvements were done. The first model built (still present in the code in q6.py) was not using dropout, and used word2vec for embedding. The accuracy result on the testing set was around 0.53, depending on the number of epochs. Adding the dropout helped reducing the over-fitting of the model on the training data. Changing the embedding method is what improved the most the results. It seems that averaging the word2vec of the words of the claims and the relevant sentences, was not relevant at all in this model. I tried to improve these embedding with normalization, but it didn't work. The main results of the final model are the summary of results given in Question 6 and can also be found in the predictions.jsonl file. As precised in question 6, the model does not predict the "predicted_evidence" field, that were left empty in the jsonl file.

Other improvements were not implemented but could have been done, such as trying Bidirectional LTSM such as in [7] because it usually learns faster. Smarter and more appropriate word embeddings methods could also be tested as the one used in [6]. Gated recurrent unit (GRU) could also be tried instead of using LSTM.

9 MISSING AUXILIARY FILES

In order to avoid duplicated computations, I cached in files the results of a lot of intermediate steps in each questions. These files are very large and therefore could not be submitted. Here is the list of all the files that were not submitted (but that are automatically recomputed if missing by the code):

- termFrequencies.json created in question 1, located in the output directory. It's a count of each word in all wikipedia articles.
- docInvertedIndex.json created in question 2, located in the output directory. It's a an inverted index of each word in wikipedia articles.
- docIdfFile.json created in question 2, located in the cache directory. It's holds the idf value of each document.
- relevant-docs.json created in question 2, located in the cache directory. It contains the set of pre-filtered relevant docs for each claim.
- claimsTfIdfFile.json created in question 2, located in the output directory. It contains the results of computations for the tf-idf value of each claim.
- docTfIdfFile.json created in question 2, located in the cache directory. It's holds the tf-idf value of each document.
- cosineSimFile.json created in question 2, located in the cache directory. It contains the final results of questions 2.
- docQueryLikelihood.txt created in question 3, located in the cache directory. It's holds the result of the queries for the claim using the models created with all types of smoothing (and without).
- fiveMostSimilarDoc.json created in question 3, located in the output directory. It contains the results of question 3.
- claimsTop5Path.json created in question 4, located in the cache directory. It contains the top5 relevant docs of claims.
- matrixifyDatasetFile.json created in question 4, located in the cache directory. It's holds the dataset used in question 4 after using the word embedding.
- testDatasetLogisticRegressionFile.json created in question 4, located in the cache directory. It contains the cached dataset used in the regression.
- testXYLogisticRegression.json created in question 4, located in the cache directory. It contains the cached test dataset used in the regression.
- testXYLogisticRegressionUnbalanced.json created in question 4, located in the cache directory. It contains the cached unbalanced test dataset used in the regression.
- embeddedTrainingTextDataPath.json created in question 6, located in the cache directory. It contains the training dataset after embedding.
- embeddedTestingTextDataPath.json created in question 6, located in the cache directory. It contains the testing dataset after embedding.
- trainingTextDataPath.json created in question 6, located in the cache directory. It contains the trainng dataset before embedding.
- testingTextDataPath.json created in question 6, located in the cache directory. It contains the testing dataset before embedding.

REFERENCES

[1] Sadia Afroz, Michael Brennan, and Rachel Greenstadt. 2012. Detecting hoaxes, frauds, and deception in writing style online. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 461–475.

[2] Yimin Chen, Niall J Conroy, and Victoria L Rubin. 2015. Misleading online content: Recognizing clickbait as false news. In *Proceedings of the 2015 ACM on Workshop on Multimodal Deception Detection*. ACM, 15–19.

[3] Giovanni Luca Ciampaglia, Prashant Shiralkar, Luis M Rocha, Johan Bollen, Filippo Menczer, and Alessandro Flammini. 2015. Computational fact checking from knowledge networks. *PloS one* 10, 6 (2015), e0128193.

[4] Niall J Conroy, Victoria L Rubin, and Yimin Chen. 2015. Automatic deception detection: Methods for finding fake news. *Proceedings of the Association for Information Science and Technology* 52, 1 (2015), 1–4.

[5] Cecilia Kang and Adam Goldman. 2016. In Washington Pizzeria Attack, Fake News Brought Real Guns. (2016). https://www.nytimes.com/

[6] Georgi Karadzhov, Preslav Nakov, Lluís Màrquez, Alberto Barrón-Cedeño, and Ivan Koychev. 2017. Fully automated fact checking using external sources. *arXiv preprint arXiv:1710.00341* (2017).

[7] Yash Kumar Lal, Dhruv Khattar, Vaibhav Kumar, Abhimanshu Mishra, Vasudeva Varma, and France Avignon. 2018. Check it out: politics and neural networks. *Cappellato et al.[5]* (2018).

[8] Chris McCormick. 2016. Google's trained Word2Vec model in Python ï£¡ Chris McCormick. (2016). http://mccormickml.com/2016/04/12/googles-pretrained-word2vec-model-in-python/

[9] Martin Potthast, Johannes Kiesel, Kevin Reinartz, Janek Bevendorff, and Benno Stein. 2017. A stylometric inquiry into hyperpartisan and fake news. *arXiv preprint arXiv:1702.05638* (2017).

[10] Hannah Rashkin, Eunsol Choi, Jin Yea Jang, Svitlana Volkova, and Yejin Choi. 2017. Truth of varying shades: Analyzing language in fake news and political fact-checking. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. 2931–2937.

[11] Victoria Rubin, Niall Conroy, Yimin Chen, and Sarah Cornwell. 2016. Fake news or truth? using satirical cues to detect potentially misleading news. In *Proceedings of the second workshop on computational approaches to deception detection*. 7–17.

[12] Kai Shu, Amy Sliva, Suhang Wang, Jiliang Tang, and Huan Liu. 2017. Fake news detection on social media: A data mining perspective. *ACM SIGKDD Explorations Newsletter* 19, 1 (2017), 22–36.

[13] Eugenio Tacchini, Gabriele Ballarin, Marco L Della Vedova, Stefano Moret, and Luca de Alfaro. 2017. Some like it hoax: Automated fake news detection in social networks. *arXiv preprint arXiv:1704.07506* (2017).

[14] Marcella Tambuscio, Giancarlo Ruffo, Alessandro Flammini, and Filippo Menczer. 2015. Fact-checking effect on viral hoaxes: A model of misinformation spread in social networks. In *Proceedings of the 24th international conference on World Wide Web*. ACM, 977–982.

[15] Andreas Vlachos and Sebastian Riedel. 2014. Fact checking: Task definition and dataset construction. In *Proceedings of the ACL 2014 Workshop on Language Technologies and Computational Social Science*. 18–22.

[16] William Yang Wang. 2017. " liar, liar pants on fire": A new benchmark dataset for fake news detection. *arXiv preprint arXiv:1705.00648* (2017).

[17] Liang Wu, Fred Morstatter, Xia Hu, and Huan Liu. 2016. Mining misinformation in social media. *Big Data in Complex and Social Networks* (2016), 123–152.

[18] You Wu, Pankaj K Agarwal, Chengkai Li, Jun Yang, and Cong Yu. 2014. Toward computational fact-checking. *Proceedings of the VLDB Endowment* 7, 7 (2014), 589–600.