ALTEGRAD challgenge: Can you predict whether two short texts have the same meaning?

Pirashanth RATNAMOGAN
MVA ENS Paris-Saclay
pirashanth.ratnamogan@ens-paris-saclay.fr

Othmane SAYEM
MVA ENS Paris Saclay
othmane.sayem@ens-paris-saclay.fr

Abstract

Like every year the Advanced learning for text and graph data class is concluded by a data challenge that should allow to use in practice what was taught during the classes. This year, the goal of this competition was to predict whether two given questions have the same meaning or not. The task is pretty complicated. Indeed Data has been manually annotated and is quite noisy because of the big ambiguity present in the comparisons.

1. Team name, member and account for submission

Our team name for this challenge is **"Ratnamogan-Sayem"**. This team is made up by **Pirashanth RATNAMOGAN** and **Othmane SAYEM**. The account used for the submission is PirashanthR. During the beginning of the competition we inconveniently submitted one submission with the account Oth Say, but then we understood that only one account has to be used to submit during the competition, so we continued using the first one only.

2. Introduction

Beyond reaching the best score possible on the kaggle competition leader-board, during this competition we have tried to reach several other goals: apply what we have seen in class and during the labs, try to apply some state-of-the-arts techniques linked to the subject. In this report we will first see the first ideas that we had developed, then we will go in details about what features and what kind of models we used to answer the problematic.

3. Pre-processing

A first thing to do in such problems, is to do a pre-processing of the data (the questions in our case) in order to "clean" it. For this task, We used basic tokenization, removed punctuation... For stopwords, we hesitated to use them since the length of the text wasn't too big and we thought it could remove some important information. Thus, we only used basic determinants (*the,a,an ...*) as stopwords. We also tried to use stemming regarding the context. After several tests, what seems to perform best was reducing the pre-processing as using only tokenization, a quite small set of stopwords and removing punctuation. We are using stemming when needed and most of the features are computed with and without stemming. Finally we have used the library `autocorrect` in order to correct the spelling of the noisy words present in the datasets.

4. First Ideas

We first tried end-to-end Deep Learning approach using an appropriate architecture (using the so-called Siamese architecture [1] . However, regarding the task difficulty and the relatively small number of example that we have, it wasn't really appropriate. Hence, we have spent most of the time of the project working on feature engineering: how to find the most relevant features in order to maximize the cross entropy.

5. Features

As described in the previous section end-to-end Deep Learning methods couldn't provide good outcomes because of the small number of training data set that we have. Deep Learning is known to automatically do the feature engineering task based on tons of examples. Here because we don't have that tons of examples we will manually create some features. The more relevant and various feature you have the best the outcome will be that's sure. We have symmetrized all the features to make the comparison of question 1 and question 2 provide the same descriptor as the comparison between question 2 and question 1 using a max/min based features order when features depends only of 1 question. We have created in total about **220 features**.

5.1. Basic Features

5.1.1 Classical NLP Features

We have created several classical NLP features for the task of document comparison. We will note q_1, q_2 the two questions that we want to compare, and Q_1, Q_2 their set of terms. For instance we have created those distances, as explained in the paper [2]:

$$\textit{Matching}(q_1, q_2) = |Q_1 \cap Q_2| \quad \textit{Dice}(q_1, q_2) = 2.\frac{|Q_1 \cap Q_2|}{|Q_1| + |Q_2|}$$

$$\textit{Jaccard}(q_1, q_2) = \frac{|Q_1 \cap Q_2|}{|Q_1 \cup Q_2|} \quad \textit{Overlap}(q_1, q_2) = \frac{|Q_1 \cap Q_2|}{\min(|Q_1|, |Q_2|)}$$

$$\textit{Cosine}(q_1, q_2) = \frac{|Q_1 \cap Q_2|}{\sqrt{|Q_1|\mathrm{x}|Q_2|}} \quad \textit{Diff}(q_1, q_2) = ||Q_1| - |Q_2||$$

$$\textit{Levenshtein}(q_1, q_2) \quad \textit{WordMoverDist}(q_1, q_2)$$

We have also used really basic features such as the difference of the number of words present in each questions, the difference of each words length, the total number of words present in the two questions, the number of un-matching words. We have also used indicator functions to detect the presence of "why","how","what","when" and "where" in each question. We have also created various features that are comparing the n-grams present in each questions. Most of the described features have been doubled, they have been computed on raw text then on the text after stemming. In order to compute **Levenshtein distance** and all its derivatives (partial,qratio ...) , that represents how much characters must be changed to go from a question to another one, we used the library `fuzzywuzzy`. To compute the classical **word movers distance** we used the pre-trained word2vec binary that we used in class and the library `gensim`.

5.1.2 Word Embedding based Features

We have created various embeddings for each question in order to create new features.
We have used conversion of each question using **Tf-Idf vectorizer** and the Raw Term Frequency vectorizer and a pre-processing with PCA (LSA features).
Using the pre-trained **Word2Vec** and **Glove** binaries and their associated embeddings. We computed a simple embedding for each question by using the mean of each word vectors using Word2Vec and Glove.
Finally we have created a more complex questions embedding using **Doc2Vec** based on `gensim` library. In our process, we are using the two types of training for Doc2Vec, creating two new embeddings (dm and dbow). Using all these embeddings we compared each question using various distances: euclidian, cosine similarity, cityblock, Jaccard, Canberra, Braycurtis and Minkowski.

5.2. Graph-of-questions based features

We have created a graph based on the comparison present in both the train and test set. Each question corresponds to a node of the graph and an edge is present between two questions if they are compared within the training or the test set. This graph allows to take advantage of the fact that if two questions are compared and that they are both also compared to another question that means that they must be quite similar.
We have extracted a lot of features for the comparison of two questions using various graph features (all the features as been symetrized in order to make the comparison question1/question2 provide the same descriptors as the comparison question2/question1). First the most basic and direct ones (from igraph library) : degrees of each node, ratio of this degree, neighbors of each node, core number of each node, common neighbors, total neighbors, various comparisons from neighbors of neighbors list (average neighbors of neighbors, dice, jaccard, overlap, cosine), number of non common neighbors, page rank score for each node, shortest path between the two nodes (after removing the edge between the two questions), from the subgraph obtained by taking the subgraph based on the neighborhood of order 3 of the 2 questions, we have extracted various features (eigenvector centralities of the two questions, edge disjoint path, page rank of each node, transitivity undirect (probability that two nodes are connected), betweenness, assortativity degree, average path length, density, eccentricity of the two nodes (maximum length of the path to another node), similarity inverse log weighted, maxflow value, similarity jaccard).

Finally, we have created a node embedding in this graph that allows to compare each question using the deep walk based approach seen in class and the `gensim` library. We have mostly take inspiration from the TP "Deep Learning from Graph". Using this embedding we have created various features using various distances: euclidian, cosine similarity, cityblock, jaccard, canberra, braycurtis and minkowski.

5.3. Graph-of-words based features

In order to measure the similarity between two pairs of questions, we decided to use the Graph kernel method as seen in the Lab 6 of the course. Using a graph-of-word approach, we transformed each sequence in the training set into a graph. Once all the graphs created, we implemented *Shortest Path Kernel* for each couple of sentences : Given a couple of questions, we created two graphs using a BoW, computed the SP kernel value of these two graphs, and used it as a feature. For instance, two similar graphs would take the value 1, and completely different graphs would take 0.

In this part, we got inspired by the article [2] which demonstrates that such approach, based on BoW and graph kernels, outperforms traditional techniques in documents similarity. The parameters of the method, such as d the maximum distance between two vertices in the graph, and the window size (equal to 2) for creating our BoW, were all inspired by the parameters in the cited paper.

5.4. Stacking to extract features from different embeddings

To extract more consistent features from word2vec and glove we have used the classical Stacking Approach. This approach allows to learn new features using models that could be used by our final model. In order to have no bias features using this approach a simple procedure have to be followed: split the data in two sets A and B, train a model using one the set A,predict the desired features for the set B using this algorithm, do the same thing one more time inverting sets A and B.
In order to compute features from what has been seen previously we have divided each features described previously into various sets (Basic Features, Graph of questions features and Graph of Word features) and we have used various models to extract features from those sets various comparisons. The basic models used to extract stacking features are: XGBoost, LGBM, Random Forest, Elastic Net, SVM (removed because too costly), linear regression, logisic regression.

Finally, the more advanced and the more efficient model used for this task is a Deep Neural Network with Siamese architec-

ture based on convolutional neural networks that will see both the manually computed features (various combinations of the previously described sets) but also each questions word embeddings using word2vec and glove. Describing such a model in details within this 3 pages report will be too long. One can check the architecture used within the code commentaries.

6. Model and architecture selection

6.1. Test of various models

In order to compare various models we will split the train dataset in two parts. One will be used to train the model (80% of the dataset) and the other one will be used to evaluate it (20% of the dataset). We will use as loss the binary cross entropy. Early stopping has been used in some of the following outcomes. For the final outcome we have used ensemble methods described in the next part. One can find the final table outcome in the Annex, LGBM and XGBoost performs the best and our Deep neural network perfoms a little less but provide and correct some outcome of the tree decisions based algorithms.

Model	Parameters	Train Loss	Test Loss
Random Forest	n_estimators=300	0.039	0.148
KNN	n_neighbors=100	0.384	0.390
Logistic regression	-	0.304	0.307
DNN	Siamese network	0.11	0.149
XGBoost	max_depth=6 n_estimators=200	0.08	0.136
LGBM	max_depth=6 n_estimators=200	0.08	0.135

6.2. Tackle overfitting

6.2.1 Using various models

In order to improve our outcome and reduce overfitting, we will create three independant models based on the best performing models: XGBoost, LGBM and Deep Neural Networks (with Siamese Network). To compute the final outcome we will average the three outcomes (that actually appears to perform best). LGBM and XGBoost are Boosting algorithms that have already a built-in regularizer, which we can tune to prevent overfitting. We tried to play with those parameters to improve the final outcome. We also used Dropout for the DNN to prevent overfitting.

6.2.2 Bagging to reduce overfitting

We have applied the famous Bagging approach in order to regularize our outcome. We are training 50 models of each of the three used algorithm (XGBoost, LGBM and Deep Neural Networks) and we will average the prediction given by those 50 models. In order to create various models, we are randomly picking 90% of the training data that are used for the training and the remaining 10% are used as validation data for early stopping.

6.2.3 Early Stopping approaches

As described above we are using 10% of the data for early stopping in each model. We used early stopping and model checkpoints approach to stop the training process when it starts overfiting (based on the validation score).

Combining those three approaches has largely allowed to improve the score,

6.3. Final Model

6.3.1 What we could have done to improve our outcome

We have implemented various other ideas that could have well improved the final outcome but we haven't succeeded to make them work efficiently during the test phase:

1. When a question A is duplicate of a question B and a question C is duplicate to A. Using the obvious present transitivity we could add to the training set that A and C must be duplicate. Moreover if D is not a duplicate of A, we could add to the training set that B and C are not duplicate of D. That method allows to improve the training database (about 300 000 examples) but it appears to lead to overfitting to the training set, we haven't succeeded to make this method work efficiently.

2. We tried to use POS Tag and Lemmatization as a preprocessing but it appears to deteriorate the resulting performance.

3. We tried various post processing to improve the final score but we haven't found anything efficient

4. We have implemented attention networks based model but it doesn't perform better than our Siamese network (an architecture is developed in the function create_neural_network_3

5. We tried to add various Graph of words features but it doesn't help to improve our final outcome (has been commented in the submitted code).

6.3.2 Brief summary & Conclusion

During this project we have tried to use various outcomes from the multiple fields that we have seen in class (Graph Mining, Deep Learning, NLP, Word Embedding, ...) and we have created more than two hundred features. We have also used a lot of regularization and ensemble methods in order to get a good score. The challenge was pretty exciting and interesting, it allowed us to learn a lot!

References

[1] S. Chopra, R. Hadsell, and Y. LeCun. Learning a similarity metric discriminatively, with application to face verification. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 539–546. IEEE, 2005.

[2] G. Nikolentzos, P. Meladianos, F. Rousseau, Y. Stavrakas, and M. Vazirgiannis. Shortest-path graph kernels for document similarity. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1891–1901, 2017.