Quora challenge

Delivery 1

Natural Language Processing

Authors

Alejandro Astruc

Joel Dieguez

Alba García

Clàudia Valverde

April 2024

Contents

1 Introduction

The goal of this project is to try to solve the Quora Challenge by applying natural language processing (NLP) advanced techniques to classify whether question pairs are duplicates or not.

We propose two solutions 1) Simple Solution and 2) Improved Solution. The simple solution acts as a baseline for comparison with the improved one. In the improved solution, we gradually introduce complexity through advanced feature engineering and embeddings to achieve the most optimal model. This iterative approach allows us to explore various levels of complexity and select the best-performing model.

The group members are Alejandro, Joel, Alba and Clàudia. Alba undertook the tasks of data cleaning, pre-processing, and developing basic features. Clàudia focused on the Simple Solution, creating distance features, exploring various vectorization techniques, and conducting the ablation study. Alejandro's primary focus lay in implementing Word2Vec, while Joel concentrated on complex sentence representations, leveraging various pre-trained LLMs. Throughout each stage, all team members trained different classifiers to identify the best-performing approach. However, only the classifier performances from approaches with promising ablation results are presented in the results section.

Our code can be found in the following github repository: `https://github.com/ClaudiaValverde/quora-challenge/tree/main`.

2 Simple solution

In the simple solution we propose is a Logistic Regression model trained on the data which was converted to numerical representation using a CountVectorizer.

	Accuracy	F1	Precision	Recall	ROC AUC
Train	0.814	0.731	0.782	0.686	0.787
Validation	0.749	0.642	0.677	0.611	0.720
Test	0.758	0.655	0.695	0.619	0.729

Table 1: Simple Solution Model Performance Metrics

While this basic approach shows decent performance (Table 1), it is clear it can be improved. Firstly, no data pre-processing was conducted; only relying on the count vectorizer for the string-to-vector conversion. Secondly, we aim to introduce more specialised features like common word counts or similarity distances.

Moreover, in the realm of NLP, more complex text data representations have

demonstrated superior efficacy for such tasks. Experimenting with advanced vectorizers could yield promising results. Thus, integrating word embeddings like Word2Vec or leveraging from pre-trained models such as DistilBert we think will help improve the performance.

Finally we will also have to check different classifiers as the logistic regressor used is quite simple.

3 Improved solution

Our idea for the improved solution has been to apply all the limitations the simple solution. Our strategy involves incremental complexity at each stage. Meaning that initially, we focus on data cleaning and preprocessing, then gradually incorporate basic features, distance measures, and subsequently progress to utilizing word embeddings, followed by embeddings from LLMs. The final solution are representations extracted from a pre-trained model for the task of question similarity in a Quora Dataset.

After every approach we see how the performance keeps improving, and we also test different classifiers, the results can be found in 3.5

3.1 Cleaning data and preprocessing

The first step for the improved solution is to clean and preprocess the data so we can delete any errors or mistakes.

When dealing with strings, we will see that they can include a lot of different characters, such us letters and numbers, but also special characters, punctuation, accents, etc. We have tried to implement functions that can be used if we need to process a simpler sentence or if we need to remove repeating symbols and words. In most cases, we use **regular expressions** (or regex) to search for an specific pattern in the sentences. Our functions are the following:

- `lowercase_sentence`: transforms the input sentence into all lowercase letters. We simply use the `.lower()` method applied to strings.
- `remove_punctuation`: removes all punctuation symbols from a given sentence, including the question mark '?'. We use the regex pattern `r'[^\w\s]'` that matches all characters that are not words nor spaces.
- `remove_accents`: removes letters with accents of a given sentence. We use the `unicode` library.
- `remove_special_characters`: similar to the two before, this function removes all special characters, including punctuation symbols, spaces, accents, etc. We use the pattern `r'[^a-zA-Z0-9]'` that matches all non

alpha numeric characters, as well as the pattern `r'[^a-zA-Z]'`, for when we also want to remove the numbers.

- `remove_stopwords`: removes stop words from a sentence, that is a set of the most commonly used words in a language. In this case we use the set `stopwords` from the library `nltk.corpus`.
- `normalize_spaces`: replaces all consecutive white space characters in the text string with a single space. In this case, we use the pattern `r'\s+'`.

See some examples of these functions below:

Original sentence:

```
Hello, höw are you doing? I hope everything is \ going well!
Lét's meet at 3:00 PM. (It's raining outside.)
```

In lower case:

```
hello, höw are you doing? i hope everything is \ going well!
lét's meet at 3:00 pm. (it's raining outside.)
```

Without punctuation symbols:

```
Hello höw are you doing I hope everything is  going well
Léts meet at 300 PM Its raining outside
```

Without accents:

```
Hello, how are you doing? I hope everything is \ going well!
Let's meet at 3:00 PM. (It's raining outside.)
```

Without special characters:

```
Hello  h w are you doing  I hope everything is   going well
L t s meet at 3 00 PM   It s raining outside
```

Without special characters nor numbers:

```
Hello  h w are you doing  I hope everything is   going well
L t s meet at      PM   It s raining outside
```

Normalised spaces after removing special characters:

```
Hello h w are you doing I hope everything is going well
L t s meet at PM It s raining outside
```

Without stop words:

```
Hello , höw ? hope everything \ going well !
Lét 's meet 3:00 PM . ( 's raining outside . )
```

Another strategy to remove errors from the data is to implement a spelling checker, for that, we have used a **BK Tree**. A BK Tree is a tree data structure used for efficiently storing and searching for data based on their similarity to a query. In our case we have used the edit distance to compare the similarity between two words. As seen in class, we build the BK Tree in the following way: first, select an arbitrary element from the dataset as the root node. Then, insert other elements into the tree according to their distance from the root node. If a new element collides with an existing node, recalculate the distance from that node.

Here it is an example of our spelling correcter function used with the set `words` from `nltk.corpus` as the vocabulary of the tree.

Original sentence:

```
the man wentt to the antimonarchik protest
because he did not like the king
```

Corrected sentence:

```
the man went to the antimonarchic protest
because he did not like the king
```

Another strategy to preprocess data is to use **Stemming and Lemmantization**. For that we have implemented the following functions:

- `stem`: given a sentence, it returns the same sentence stemmed. That is, it removes the prefixes and suffixes of every word of the sentence. Note that it can produce words that do not actually exist. We have implemented both the Porter Stemmer and the Lancaster Stemmer. The first, is more conservative whereas the second is a more aggressive stemmer and produces shorter stems.
- `lemma`: given a sentence, it returns the base word (or lemma) for each word of the sentence. In this case, the outputs are valid words. We use the `WordNetLemmatizer` from `nltk.stem`.

See some examples of these functions below:

Original sentence:

```
Hello, höw are you doing? I hope everything is \ going well!
Lét's meet at 3:00 PM. (It's raining outside.)
```

Stemmed sentence (Porter):

```
hello , höw are you do ? i hope everyth is \ go well !
lét 's meet at 3:00 pm . ( it 's rain outsid . )
```

Stemmed sentence (Lancaster):

```
hello , höw ar you doing ? i hop everyth is \ going wel !
lét 's meet at 3:00 pm . ( it 's rain outsid . )
```

Lemmantization sentence:

```
Hello , höw are you doing ? I hope everything is \ going well !
Lét 's meet at 3:00 PM . ( It 's raining outside . )
```

In the final result, we have not used the BK Tree, nor the Stemming nor the Lemmantization. That is because they take too much computation time, and we already found an improved result with the other approaches.

3.2 Feature Engineering

Our next idea is to extract useful features from the data so that we can improve the performance of our base solution.

3.2.1 Basic text features

Having in mind our particular problem, we have thought of functions that, when applied to our questions, can produce a meaningful and useful result to compare them. In this case, we have the following:

- `number_words`: returns the number of words in a sentence. It tokenizes the sentence with the `word_tokenize` from the `nltk` library.
- `number_common_words`: returns the number of common words between two given sentences. It also uses the same tokenizer as before.
- `number_common_words_2`: similarly, it returns the number of common words that are in the same position between two given sentences.
- `first_word_equal`: returns either '1' or '0' if the first word of two given sentences is equal or not.

- `last_word_equal`: returns either '1' or '0' if the last word of two given sentences is equal or not.

Adding only these five features to the simple solution with the count vectorizer and the logistic model, we already see an improvement of the performance by **3%-5%**. With that, we concluded that the features are useful and we will keep them in or more advance solution. Later, we will also test different combinations for all approaches with different classifiers.

3.2.2 Distance features

We also have computed features based on different type of distances in order to have a measure of similarity between the two questions. In this section we explain what distances have been computed.

- `edit_distance`: its a string metric, i.e. a way of quantifying how dissimilar two strings (in our case the paired questions) are to one another, that is measured by counting the minimum number of operations required to transform one string into the other. We implemented this function implemented with `cython`, as it requires different initialization of the function, in the `util.py` file its commented and acutally implemented in `utils_claudia.ipynb`.
- `jaccard_similarity`: also known as Jaccard index, is a statistic to measure the similarity between two data sets. It is measured as the size of the intersection of two sets divided by the size of their union. Jaccard similarity can be used in natural language processing to compare texts, text samples, or even individual words.
- `cosine_similarity`: Given two vectors, the cosine similarity is the dot product between them divided by the norm of both of them. The result is always between 1 and -1, 1 meaning that the vectors are the same and -1 that the vectors are opposed. Another peculiarity of this measure is that it does not take into account the size of the vector, only its direction.
- `euclidean_distance`: It is the distance between two point in an euclidean space (such as $\mathbb{R}^n$). In an embedding space, it can be used to represent how close two embeddings are. It is calculated as the euclidean norm of the difference of the two points.

3.2.3 Advanced encoders

- **Count Vectorizer**: Implemented in the simple solution it converts a collection of text documents (questions in our case) to a matrix of token counts. It breaks down each document into words or n-grams. Then, it

counts the occurrences of each token in each question. Finally, it converts the token counts into a matrix representation, where each row corresponds to a question and each column corresponds to a token, with the cell value representing the count of that token in the question.

We used it for the ablation study in order to see if the basic features and distance features created where giving further information to the model.

- **TF-IDF**: Term Frequency-Inverse Document Frequency, is a measure of the importance of a word relative to a sentence in a corpus.
- **Feature Hash**: Implemented with `sklearn`, the class `FeatureHasher` is a high-speed, low-memory vectorizer that uses a technique known as feature hashing, or the "hashing trick". Instead of building a hash table of the features encountered in training, as the vectorizers do, instances of `FeatureHasher` apply a hash function to the features to determine their column index in sample matrices directly. Despite having implemented it, this vectorization technique showed poor results since the beginning. We have opted to exclude it from further experiments.

3.3 Implementation of embeddings

3.3.1 Word embeddings

Another tool we tried is word embeddings. The idea is that from a word embedding we can try to derive a sentence embedding for each question. We used the `gemseim` library and its `Word2Vec` model implementation. For this purpose we cleaned the data and trained a Word2Vec model. Some relevant functions we created along the way are:

- `preproces_data_Word2Vect`: This is a cleaning and preprocessing function that lemmatizes and discards non alphanumeric words, among other tasks.
- `get_Word2Vect_from_clean`: This function receives the dataframe of questions and returns their vector embeddings. It calls to `doc_to_vec(sentence, word2vec)` for each sentence.
- `doc_to_vec`: Is the function that uses word embeddings to generate sentence embeddings. In our case we perform the average of the vectors of each word within a sentence. This is the simplest approach and can be further improved upon.

For vectorizing our data we resorted to two different `Word2Vec` models:

1. First we tried training our own `Word2Vec` model on the data after cleaning and preprocessing.

2. Then we used a pre-trained `Word2Vec` model: glove-wiki-gigaword-300 from the `gemseim` library.

In both cases the embedding dimension for each sentence was 300 resulting in a 600 dimension vector for each pair of sentences. We then proceeded to train two logistic regressors on the two separate embeddings generated by each `Word2Vec` model for further comparison. We obtained similar results with both of them.

One of the main drawbacks of this strategy is that many infrequent words, or words that are not in English are lost in the vectorization process, loosing relevant information along the way. Moreover we encountered a series of problems and errors during implementation, for example, we had to introduce an extra check in `doc_to_vec(sentence, word2vec)` due to the fact that the `Word2Vec` model we trained had keys for indexes that were bizarely out of bounds of the list of vectors it had generated as embeddings.

3.3.2 Advanced embeddings

In another line of study, we decided to research the current literature on the matter and found whole sentence embeddings using BERT transformers. The library `Sentence-Transformers` provides trained transformers for different goals. In our case, we focused in the Sentence Transformer `all-MiniLM-L6-v2`. We chose this model because it was of the ones with best results from the library and, at the same time, it is very fast and small. It is based on the MiniLM architecture, very similar to BERT but with much less parameters.

Some specifications of the model are: It has less than 33M parameters (we could not find the exact number), divided in 6 attention layers, and for pooling, it uses Mean Pooling. It is very small, only 80MB. It takes as input sentences of max length 256 and outputs their embeddings in a 384 dimension space. We used its pretrained version, trained in over one billion training pairs.

Once we chose the model, we decided to test its performance in two different ways, calculating the similarity between embeddings and using a classifier over the embeddings.

On the one hand, we considered to evaluate the **similarities between embeddings** with the following approaches: Dot product, Cosine Similarity and Euclidean Distance. Dot product was a bit lacklustre and Cosine and Euclidean had very similar results. All of them were tested using just the raw scores and using a Logistic Regression classifier with just the similarity values. In the end, we decided the best one was the Cosine Similarity because it had better mathematical properties, thus we tested it with other classifiers as well. All of them performed very similar, as they had only one float as input, so did not consider it relevant to add the results here.

On the other hand, we used just the **embeddings** as input for the classifiers, collecting 768 features in total per pair. The results here vary a lot between

classifiers.

3.4 Leveraging from a pre-trained model on a quora question similarity task

Finally, we decided to also use a pre-trained Cross-Encoder from the same library `Sentence-Transformers`. It takes two texts as inuts and outputs a score from 0 to 1 according to how similar they are. This architecture is composed by a Transformer Encoder like BERT and a classifier. Similar to our last approach but it's all together.

In particular, they have a set of models pretrained on a very similar problem in Quora Duplicate Questions. We chose specifically the model `quora-distilroberta-base`, which uses a Distil Roberta base. As you can see in the Results tables (3.5.3), this model had the best results overall, even without fine-tuning.

3.5 Model selection

In this section we explained the different studies we have performed in order to obtain the best representation of features as well as the best classifier model.

3.5.1 Ablation study

With this study, our intend has been to see how the performance of the classifier improves by adding complexity to the representations of the questions.

Approach	Accuracy	F1	Precision	Recall	ROC AUC
Simple Solution	0.749	0.642	0.677	0.611	0.720
CountVectorizer+BF	0.780	0.693	0.714	0.673	0.757
TF-IDF+BF	0.781	0.690	0.726	0.657	0.756
TF-IDF+BF+DF	**0.790**	0.702	**0.738**	0.668	0.765
Word2Vec	0.671	0.390	0.630	0.283	0.592
Word2Vec (gemseim)	0.689	0.461	0.624	0.365	0.620
Sentence Similarity	0.779	**0.707**	0.692	**0.723**	**0.767**
Sentence Embeddings	0.716	0.548	0.662	0.468	0.664

Table 2: **Ablation Study Performance Metrics.** BF: Basic Features, DF: Distance Features, reporting validation metrics for all with the classifier that performed the best.

As we can see from the ablation study (2), adding distance information gives more information to the model than by just having basic features. TF-IDF has also proven to be a slightly better vectorizer. Between the embeddings,

to our sourprise Word2Vec does not perform as good as we hypothesized it would, but sentence similarity with advanced embeddings has also one of the best performances.

Our approaches that work best are, in one hand, TF-IDF vectorizer with basic features and distance features (TF-IDF+BF+DF), and on the other hand, Sentence Embeddings classifing by similarity measures.

3.5.2 Model selection study

We have also done a study for choosing the best performing classifier, we have tested different Logistic Regressor, XGBoost and Multi Nomial Naive Bayes, Random Forest and LGBM, for some of the approaches some of the classifiers have been disregarded as we encountered some incompatibilities.

Because with the ablation study (2) we have obtained two different approaches that perform very similar, we have decided to choose the most optimal classifier for both approaches. For the advanced embeddings approach 3.3.2, we have conducted this experiment for both, sentence classification by similarities and embeddings as for the different classifiers their performance oscilates the most and we would be disregarding one of the best resutls (3, 4, 5).

Classifier	Accuracy	F1	Precision	Recall	ROC AUC
Logistic Regressor	0.790	0.702	**0.738**	0.670	0.766
XGBoost	**0.793**	**0.715**	0.726	**0.705**	**0.775**
MultiNomialNB	0.689	0.615	0.566	0.673	0.686

Table 3: **Classifier Selection for TF-IDF+BF+DF Approach Performance Metrics.** BF: Basic Features, DF: Distance Features, reporting validation metrics

Classifier	Accuracy	F1	Precision	Recall	ROC AUC
Logistic Regressor	**0.779**	0.707	**0.692**	0.723	0.767
Random Forest	**0.779**	**0.710**	0.688	0.733	0.769
XGBoost	0.778	0.707	0.690	0.726	0.767
LGBM	0.778	**0.710**	0.684	**0.738**	**0.770**

Table 4: **Classifier Selection for Sentence Similarity Approach Performance Metrics.**

Looking at the performance of the classifiers in the TF-IDF+BF+DF approach (3) we can conclude that the best performing one is XGBoost. Sentene Similarity (4) has very consistent performance accross the different classifiers, but for further comparisons, we have chosen Logistic Regressior for further comparisons. On the contrary, Sentence Embedding (5) approach has more variety in

Classifier	Accuracy	F1	Precision	Recall	ROC AUC
Logistic Regressor	0.715	0.548	0.662	0.468	0.664
Random Forest	0.650	0.100	**0.971**	0.053	0.053
XGBoost	**0.837**	**0.780**	0.774	**0.787**	**0.826**
LGBM	0.821	0.752	0.772	0.733	0.803

Table 5: **Classifier Selection for Sentence Embeddings Approach Performance Metrics.**

the performance of the classifier, having more basic classifiers as Logistic Regression and Random Forest performing quite bad, but the more complex ones (XGBoost and LGBM) performing very well, we think this is probably due to the fact that the embeddings have a more complex representation of the questions but also give more context and thus are more explicative for classification.

3.5.3 Final models results

In this section we aim to compare the different approaches with their corresponding best classifier. We have selected four approaches as the best ones:

1) TF-IDF+BF+DF,

2) Sentence Similarity,

3) Sentence Embedding,

4) Pretrained model on Quora question similarity task.

As it can be seen in the comparison table for the results (6), as we increase the complexity of the representations of the questions (from basic features, going through distances and finally complex embeddings), the accuracy and overall performance of the models increases, ending with the best model which is a pre-trained Cross-Encoder model on a very similar task as our challenge.

Approach	Classifier	Accuracy	F1	Precision	Recall	ROC AUC
1)	XGBoost	0.795	0.720	0.730	0.710	0.777
2)	Log Regressor	0.783	0.714	0.701	0.727	0.772
3)	XGBoost	0.840	0.785	0.784	0.786	0.829
4)	CrossEncoder	**0.918**	**0.887**	**0.899**	**0.876**	**0.909**

Table 6: **Final results for the best approaches.** BF: Basic Features, DF: Distance Features, LR: Logistic Regressor, reporting test metrics for all with the classifier that performed the best.

4 Conclusions

In conclusion, we are happy with the results achieved in this project. Although having found an open-source pre-trained model for our specific task has been of great help to have very good results, we believe our approach of gradually increasing complexity in pursuit of the optimal sentences representations and classifier proved effective. This has also helped us to incorporate everything learned in class, from employing feature egeniering with more basic NLP techniques, create basic embeddings with Word2Vec and conducting experiments such as ablation studies and testing simple classifiers, we gained valuable insights.

Some further work we would have like to do was to combine both Word2Vec and advanced embeddings with both basic and distance features. Additionally, training a transformer model from scratch was something we wished to explore. However, constraints in terms of time and computational resources prevented us from pursuing this endeavour. Despite these limitations, we are satisfied with the progress made and the knowledge gained from this project.