Universitat de Barcelona

Facultat de Matemàtiques i Informàtica

Natural Language Processing

First project

Kaggle Quora Challenge

Sara Bardají Serra

Sergi Bech Sala

Àlex Pujol Vidal

David Rosado Rodríguez

Jordi Segura Pons

Barcelona, April 26, 2023

1 Delivery comments

Regarding the creation of the conda environment, notice that we found an error when creating environment using the requirements.txt. Run these commands if you are running the channels error:

```
conda config –env –add channels conda-forge
conda create –name quora_test_env –file requirements.txt
```

Regarding the delivery format, notice that we have added one additional folders to the zip, *cython_utils*, in which we add necessary files for compiling and creating functions in cython.

2 Text preprocessing

In this section, we will explain the text processing we have carried out before obtaining features and training a classifier.

2.1 Text Normalization

Text normalization refers to the process of transforming raw text data into a standardized form, which can include tasks such as converting all text to lowercase, replacing contractions with their expanded form, and replacing common abbreviations or acronyms with their full form.

We first tried to do a Name Entity Regonition (NER) in order to remove, or substitute, different name entities by common words. For example, "John" would be be substituted by "PERSON". Here we have the entities we chose:

- **PERSON**: Refers to names of people, which may be important in questions and answers that involve people or personalities.
- **GPE** (Geo-Political Entity): Refers to names of countries, cities, and other geopolitical entities, which may be important in questions and answers that involve locations or politics.
- **LOC** (Location): Refers to other location names, which may be important in questions and answers that involve places or travel.
- **DATE**: Refers to dates, which may be important in questions and answers that involve historical events, schedules, or timeframes.
- **TIME**: Refers to times, which may be important in questions and answers that involve schedules or specific moments.
- **MONEY**: Refers to monetary values, which may be important in questions and answers that involve finance or pricing.
- **ORG**: Refers to names of organizations or companies, which may be important in questions and answers that involve business or industries.

After that, we unified contractions and abreviations, so for example “we can’t go...” it is changed by a “we cannot go...” in this way we achieve a more normalized text. Nevertheless, the NER normalization has not been added to the final model as we found that we might lose relevant information to our task.

2.2 Text Cleaning

Text cleaning is the process of preparing text data for analysis by removing or modifying any unwanted or irrelevant information, which can include tasks such as removing punctuation symbols, tokenization, remove stop words, normalize spaces, and the treatment of special tokens.

Tokenization is the process of breaking down a text document into smaller units called tokens. The “tokenize_text” function converts all tokens to lowercase, making them more uniform and easier to analyze.

Punctuation, non-alphabetic characters, and English stopwords are often not informative, the functions in this notebook, such as “remove_punctuation,” “remove_nonAlphaWord,” and “remove_stopwords,” remove these unwanted characters from the text data to make it cleaner and more manageable. Regarding stopwords, we have first tried to remove all stopwords provided by the nltk package, but we observe that too many important words were removed, words as how, where, which, etc. that are crucial to our task. Hence, we decided to create a little list of the stopwords that we decided to remove.

Normalization of spaces is another crucial task in text cleaning, and the ”normalize_spaces” replaces consecutive whitespace characters in the text string with a single space, making the text more uniform and easier to analyze.

Finally, the “remove_accents” function and “special_tokens” function perform additional text cleaning tasks. The “remove_accents” function replaces Unicode characters with their closest ASCII equivalents, making the text more readable and uniform. The “special_tokens” function replaces rare words (words that only appear once) with a special token, making the text more manageable and easier to analyze.

2.3 Spell Checker

The spell checker uses a **BKTree** class, which is a data structure for efficiently searching a set of words for words that are close to a query word based on their edit distance. Edit distance is a measure of the similarity between two strings. The smaller the edit distance, the more similar the strings are.

The **spellchecker** function takes a query text and a vocabulary set as input and returns a corrected version of the query text. For each word in the query, it checks if it exists in the vocabulary set. If not, it uses the BKTree to search for the closest word in the vocabulary within a specified edit distance (in this case, 2). The corrected word is then appended to the final corrected text.

Unfortunately, we were not able to use the spellchecker due to its high computational cost.

3 Feature engineering

Once the text has been processed, we are ready to obtain some features on the corpus. Let us make a list with all the feature ideas we have had, although not all of them have been used in the end. We thought it would be interesting to implement all of them and try then try different combination to see which provided a better performance. A more thorough explanation of each of these features can be found in the individual notebooks.

- **words_count**: Counts the number of words in a given text.
- **nonAscii_word_count**: Counts the number of non-ASCII words in a given text.
- **one_hot_begin**: Creates a one-hot encoding for the first word in each question of the corpus.
- **first_word_equal**: Computes whether the first word of two questions is equal.
- **last_word_equal**: Computes whether the last word of two questions is equal.
- **common_words_count**: Computes the number of common words between two questions.
- **common_words_ratio**: Computes the ratio of common words between two questions to the total number of words in both questions.
- **fuzz_ratio**: Computes the fuzzy string matching ratio between two questions.
- **longest_substring_ratio**: Computes the ratio of the length of the longest common substring between two questions to the length of the shorter question.
- **num_of_characters**: Counts the number of characters in a text.
- **difference_word_count**: Computes the absolute difference in the word count between two texts.
- **num_of_unique_words**: Counts the number of unique words in two concatenated texts.
- **num_of_words**: Counts the total number of words in two concatenated texts.
- **total_unique_words_ratio**: Computes the ratio of the number of unique words to the total number of words in two concatenated texts.
- **oov_count**: Computes the number of out-of-vocabulary words in a text, given a vocabulary.
- **rare_word_count**: Computes the count of rare words in a text, given a word count dictionary and a threshold for rarity.
- **named_entity_overlap**: Computes the named entity overlap between two texts.
- **compute_word2vec_embeddings**: Computes the word2vec embedding for a given text by taking the mean of embeddings of all words in the text.

- **compute_minkowski_similarity**: Computes the minkoswki similarity between two given word embeddings.
- **compute_cosine_similarity**: Computes the cosine similarity between two given word embeddings.
- **compute_euclidian_similarity**: Computes the euclidian similarity between two given word embeddings.
- **compute_cityblock_similarity**: Computes the cityblock similarity between two given word embeddings.
- **compute_fasttext_embeddings**: Computes the FastText embedding for a given text by taking the mean of embeddings of all words in the text.
- **count_word_syllables**: Counts the number of syllables in a tokenized word from a sentence.
- **count_sentence_syllables**: Counts the number of syllables in an entire sentence.
- **Flesch_Reading_Ease**: Computes the Flesch Reading-Ease score of a sentence. The Flesch Reading-Ease is a readability test that measures how easy a text is to read. It considers the number of words in a sentence and the number of syllables in a word. The output is a score that usually ranges from 0 to 100, with higher scores indicating easier-to-read text.
- **Flesch_Grade_Level**: Computes the Flesch-Kincaid Grade Level of a sentence. The Flesch-Kincaid Grade Level is another readability metric that estimates the education level required to comprehend a text easily. The result is expressed as a U.S. grade level, which means that a text with a score of 8.0, for example, should be easily understood by someone with an 8th-grade education level.

Moreover, we have also built a tf-idf from scracth. We first try a simple python implementation of the tf-idf, but given its computational cost, we thought it would be a good idea to speed up its compilation and move it to cython. The implementation can be found on the folder *cython_utils*. An example of use can be found on *utils_AlexPujol*. The improvement in speed was astonishing!

4 Models

4.1 Simple solution

In a first attempt to solve the problem, we build a simple model for which we use a CountVectorizer to convert the questions into numerical data, with no previous preprocessing, and without any other features we train a Logistic regression model.

The results obtained with this model can be seen in Table 1.

	Accuracy	Precision	Recall	ROC-AUC	Log Loss
Training	0.814	0.782	0.687	0.788	0.409
Validation	0.749	0.677	0.610	0.720	0.518
Test	0.758	0.695	0.619	0.729	0.508

Table 1: Results obtained with the simple approach.

4.2 Improved solution

In an attempt to improve the previous solution, we decided to preprocess the questions by applying some text normalization techniques and some text cleaning. This preprocessing has been explained in detail in Section 2. Although we did implement a spellchecker, in the end we did not use it in our preprocessing because of the large amount of time needed to run it on the question dataset.

After the text preprocessing, we performed some feature engineering. We tried to add features that we believe could help the model to better distinguish if two questions are the same or not. The final group of features we used is: num_of_unique_words, difference_word_count, common_words_count, common_words_ratio, first_word_equal, last_word_equal, fuzz_ratio, num_of_characters, total_unique_words_ratio, rare_word_count, count_sentence_ syllables, Flesch_Reading_Ease and Flesch_Grade_Level.

With these features, the fasttext embeddings and some similarites measures as minkowsky, cityblock, euclidian and cosine similarities, we trained an XGBoost model. The results can be seen in Table 2.

We also try to incorporate other vectorization methods such as CountVectorizer and TfidfVectorizer. We trained two different models adding each of these techniques, finding that neither of these models outperformed the previous one. The results can also be seen in Table 2.

Model	Features	Acc.	Prec.	Recall	ROC-AUC	Log Loss
XGBoost	F. Eng. + fasttext	0.805	0.740	0.729	0.790	0.393
XGBoost	F. Eng. + CV	0.802	0.737	0.720	0.785	0.397
XGBoost	F. Eng. + TF-IDF	0.802	0.741	0.714	0.784	0.395

Table 2: Results in the validation set using different features and an XGBoost classifier.

4.2.1 Top features with different classifiers

At that point, we decided to continue using only the engineering features and the fasttext embeddings. We use the 300-top features obtained by XGBoost in order to halve the number of features used. With the selected features we have trained three different models: an XGBoost, a RandomForest and an Histogram GradientBoosting. Furthermore we have also performed an ensemble method with the XGBoost and the Histogram GradientBoosting, in an attempt to increase even more the performance. Notice that the best result is achieved with the Histogram GradientBoosting, reaching a log loss of 0.383 in the validation set. A comparison of the results obtained with these models can be found in Table 3.

Model	**Features**	**Acc.**	**Prec.**	**Recall**	**ROC-AUC**	**Log Loss**
XGBoost	Top	0.801	0.732	0.725	0.785	0.398
Random Forest	Top	0.718	0.685	0.437	0.660	0.532
Hist-GradientBoosting	Top	0.813	0.752	0.735	0.797	0.383
Ensemble	Top	0.811	0.749	0.735	0.795	0.385

Table 3: Results in the validation set using the top feature obtained from the XGBoost model and trained on different classifiers.

5 Conclusions

In this project, we explored various methods to determine if two questions are duplicates. We began with a simple solution using a CountVectorizer and a Logistic Regression model as a baseline, which achieved modest results. To improve upon this, we implemented text preprocessing and feature engineering techniques, which allowed us to create a more sophisticated model using FastText embeddings and additional features, such as Minkowski, cityblock, Euclidean, and cosine similarities.

We experimented with different vectorization methods, including CountVectorizer and TfidfVectorizer done in Cython, but ultimately found that FastText embeddings provided the best results when combined with our engineered features. We then trained multiple classifiers, including XGBoost, Random Forest, and Histogram GradientBoosting, using the top 300 features identified by the XGBoost model.

The best performance was achieved by the Histogram GradientBoosting model, which demonstrated a log loss of 0.383 in the validation set. We also experimented with an ensemble method combining XGBoost and Histogram GradientBoosting, which achieved a slightly lower performance than the standalone Histogram GradientBoosting model.