Natural Language Processing

master in fundamental principles of data science

Assignment 1

Quora challenge

Authors
Irene Bonafonte Pardàs
Otis Carpay
Vladislav Nikolov Vasilev

Faculty of Mathematics and Computer Science

Academic year 2021-2022

Contents

1 Introduction

In this assignment we are going to try to solve the Quora Question Pairs challenge. Given a pair of questions, we have to automatically determine whether they are semantically equivalent or not. The goal of this is to reduce the number of duplicate questions and improve the overall user experience.

In order to solve this challenge, we are fist going to try a simple solution which will allow us to get a better understanding of the problem and identify possible flaws. After that, we are going to refine this initial solution in hopes of obtaining a model that is more robust and better able to identify duplicate questions.

2 Simple solution

As a simple solution, we are going to train a logistic regression classifier in order to detect duplicate questions. Since we cannot feed text data directly into the model, we have to use some other kind of numerical representation. In this case, we can use the bag-of-words representation in order to encode the questions.

When following this approach, we have run into some technical problems. One of them is that not every question is represented as a string. This has forced us to encode each one of the questions properly before trying to get the bag-of-words representation.

Because this approach is quite simple, it has some inherent limitations:

- No text preprocessing is applied, apart from the basic preprocessing that the `CountVectorizer` class from `scikit-learn` performs. This means that the corpus is filled with misspelled words, words spelled in different ways (for example, *e-mail* and *email*) and stop words that are not quite relevant.

- Even though the bag-of-words representation allows us to encode the questions using numerical values, it ends up falling short because it only considers the word frequency inside the document. For instance, it could also consider how many time a word appears in the whole corpus, which might be important when trying to identify relevant words. Also, there might be better ways to encode words, like using embeddings and combining them in some kind of fashion in order to create sentence embeddings.

- Using only the bag-of-words representation may not be enough. We can try to create custom metrics that allow us to capture the distance between the questions and use them in the training process, whether it is a custom metric or some kind of metric that allows us to capture the semantic difference between the sentences. Then, we can either use this metrics on their own or combine them with some other kind of representation.

3 Improved solution

In order to improve our model and methodology, we are going to do a series of changes to the simple solution that we already have:

1. We are going to apply some text preprocessing in order to remove some unnecessary tokens.
2. We are going to create new feature vectors and distances, which will allow us to better represent the text.
3. We are going to try different models to see how each one of of them performs.

3.1 Preprocessing the text (Otis)

After short experimentation with different steps of preprocessing the text, we realized that the sklearn's CountVectorizer already contains a powerful preprocessor that can be accessed using the function build_analyzer(). Besides tokenization, the preprocessor converts words to lowercase and removes very frequent and unfrequent words that are not useful for the model. The main limitation of this preprocessing is that no spell checking is performed. We have also explored the use of lemmatization but it is not very useful with word2vec embeddings, because it contains word inflexions in its vocabulary, and therefore we decided to exclude it from our models.

3.2 Creating new feature vectors and distances

Next, let's talk about the new feature vectors and distances that we have implemented. Each one of us has worked on different features, so we are going to briefly talk about our work in the next sections.

3.2.1 Vladislav's features

I have implemented two new feature vectors and a distance. The first feature vector is the **tf-idf**, for which I created a custom class that is able to learn the vocabulary and the idf values of a corpus. Later on, it will use this information to transform the input documents to sparse matrices containing the tf-idf values. The code can be seen below:

```
class TfIdfCustomVectorizer(BaseEstimator, TransformerMixin):
    def _fit(self, X):
        n_docs = len(X)

        i = 0
        self.vocabulary = {}
        word_counts = defaultdict(int)

        for doc in X:
            words_in_document = set()

            for word in doc:
                if word not in self.vocabulary:
                    self.vocabulary[word] = i
                    i += 1

                words_in_document.add(word)

            for word in words_in_document:
                word_counts[word] += 1

        word_count_array = np.zeros(len(self.vocabulary))

        for word, idx in self.vocabulary.items():
            word_count_array[idx] = word_counts[word]

        self.idf = np.log((n_docs) / (1 + word_count_array)) + 1


    def fit(self, X):
        X_processed = preprocess(X)
        self._fit(X_processed)

        return self


    def transform(self, X):
        X_preprocessed = preprocess(X)

        data, ind_col, ind_ptr = bag_of_words(
            X_preprocessed,
            self.vocabulary
        )

        for i in range(len(ind_ptr) - 1):
            current_idx, next_idx = ind_ptr[i], ind_ptr[i+1]
            cols = ind_col[current_idx:next_idx]

            bow_doc = data[current_idx:next_idx]
            bow_doc_tfidf = bow_doc * self.idf[cols]

            doc_norm = np.sqrt(np.dot(bow_doc_tfidf, bow_doc_tfidf))
            bow_doc_norm = bow_doc_tfidf / doc_norm

            data[current_idx:next_idx] = bow_doc_norm

        X_transformed = sp.sparse.csr_matrix(
            (data, ind_col, ind_ptr),
            shape=(len(X), len(self.vocabulary))
        )

        return X_transformed
```

```python


    def fit_transform(self, X):
        self.fit(X)
        X_transformed = self.transform(X)

        return X_transformed
```

As we can see, this class inherits from two base classes from `scikit-learn` and implements the same methods them. This means that it could be used in a pipeline without any kind of problems. One important thing that I would like to remark is that the formula for the idf value that I have used is the smoothed inverse document frequency, which is the following:

$$\text{idf} = \log\left(\frac{|X|}{1 + |X_w|}\right) + 1 \tag{1}$$

where $|X|$ is the corpus size and $|X_w|$ is the number of documents containing the word w. This version of the formula is always greater than zero. The formula that `scikit-learn` uses is similar to this one, but adding an extra document to the numerator.

In order to create the bag-of-words representation of the text, I have used the following function:

```python
def bag_of_words(documents, vocabulary, normalize=False):
    data = []
    ind_col = []
    ind_ptr = [0]

    for doc in documents:
        bow_doc = defaultdict(int)

        for word in doc:
            if word in vocabulary:
                bow_doc[word] += 1

        bow_array = np.array(list(bow_doc.values()))
        bow_norm = np.sum(bow_array) if normalize else 1.0

        bow_doc_normalized = [
            bow_doc[word] / bow_norm
            for word in bow_doc.keys()
        ]

        cols = [vocabulary[word] for word in bow_doc.keys()]

        data.extend(bow_doc_normalized)
        ind_col.extend(cols)
        ind_ptr.append(len(ind_col))

    return np.array(data), np.array(ind_col), np.array(ind_ptr)
```

This function can also create the normalized version of the bag-of-words representation, which means that the sum of the frequencies of a given document is one. This will become very useful later on.

The second feature vector that I implemented is a combination of the previous tf-idf representation and word embeddings. The idea is to compute both the the tf-idf values and the embedding vectors of the words in a document. Then, we weight these vectors by multiplying them by the corresponding tf-idf values and we sum them, thus creating an embedding of the sentence. I believe that this embedding will retain some semantic information of the input sentence, and can be later on used to compute other distances. Also, I believe that it makes sense that the embedding of a sentence is computed as the combination of the embeddings of the individual words.

Like in the previous case, this has been implemented as a class that can learn the necessary information and then transform the input text. The implementation of this class is the following:

```
class TfIdfEmbeddingVectorizer(TfIdfCustomVectorizer):
    def __init__(self, embeddings_type=None):
        super().__init__()

        self.embeddings_type = embeddings_type

        # Load embeddings
        if self.embeddings_type is None:
            self.model = api.load('word2vec-google-news-300')
        else:
            self.model = KeyedVectors.load(self.embeddings_type)


    def fit(self, X):
        X_preprocessed = preprocess(X)

        # Generate vocabulary and idf values
        self._fit(X_preprocessed)

        # Generator used for indexing
        def index_generator(max_idx):
            idx = 0

            while idx < max_idx:
                yield idx
                idx += 1

        reindexer = index_generator(len(self.vocabulary))

        idf_valid_idx = [
            self.vocabulary[word]
            for word in self.vocabulary.keys()
            if word in self.model.key_to_index
        ]

```

```python
        self.idf = self.idf[idf_valid_idx]

        self.vocabulary = {
            word: next(reindexer)
            for word in self.vocabulary.keys()
            if word in self.model.key_to_index
        }

        return self


    def transform(self, X):
        X_preprocessed = preprocess(X)
        X_transformed = []

        for doc in X_preprocessed:
            bow_doc = defaultdict(float)

            for word in doc:
                if word in self.vocabulary:
                    idx = self.vocabulary[word]
                    bow_doc[word] += self.idf[idx]

            bow_array = np.array(list(bow_doc.values()))
            bow_norm = np.sqrt(np.dot(bow_array, bow_array))

            bow_doc_normalized = {
                word: bow_doc[word] / bow_norm
                for word in bow_doc.keys()
            }

            doc_embedding = np.zeros(self.model.vector_size)

            for word, tfidf in bow_doc_normalized.items():
                doc_embedding = doc_embedding + self.model[word] *
    tfidf

            X_transformed.append(doc_embedding)

        X_transformed = np.array(X_transformed)

        return X_transformed
```

We can see that this class inherits from the one that we previously defined. This is due the fact that it uses almost the same information (vocabulary and idf values). We only need to post-process the vocabulary and the idf values removing the ones that don't have an actual embedding. For storing, getting and using the embeddings, we have decided to use the `gensim` package, which contains pretrained models and allows us to create our own embeddings.

Finally, I implemented the Word Mover's Distance (WMD)[1], which is an instance of the Earth Mover's Distance (EMD) for the task of computing distances between documents. Very simply put, it computes the distance that the embeddings of the words of a document have to be moved in order to reach the embeddings of the words of another document. Because the embeddings can retain the semantic information of the words, two words that have similar meanings are expected to be close. Using this distance, we could get small values for similar questions, whereas the distance for different questions is expected to be large (although this may not always be true).

The implementation is heavily inspired by `gensim`'s implementation of the WMD[1]. I did some modifications to it so that it is compatible with our previously defined functions and it better suits our problem and data structures. In order to solve the EMD problem, I used the solver provided by `pyemd`, because coding one from scratch would suppose to be very tedious, inefficient, error-prone and difficult to be properly tested. The implementation of the WMD can be seen below:

```python
def word_movers_distance_document_pair(doc1, doc2, model):
    doc1_tokens = [token for token in set(doc1) if token in model.key_to_index]
    doc2_tokens = [token for token in set(doc2) if token in model.key_to_index]

    len_tokens_doc1 = len(doc1_tokens)
    len_tokens_doc2 = len(doc2_tokens)

    if len_tokens_doc1 == 0 or len_tokens_doc2 == 0:
        return 0

    vocabulary = {
        word: idx
        for idx, word in enumerate(list(set(doc1_tokens) | set(doc2_tokens)))
    }

    doc1_idx = [vocabulary[word] for word in doc1_tokens]
    doc2_idx = [vocabulary[word] for word in doc2_tokens]

    doc1_embeddings = np.array([model[word] for word in doc1_tokens])
    doc2_embeddings = np.array([model[word] for word in doc2_tokens])

    doc1_embeddings = np.repeat(doc1_embeddings, len_tokens_doc2, axis=0)
    doc2_embeddings = np.tile(
        doc2_embeddings.reshape(-1,),
        len_tokens_doc1
    ).reshape(-1, model.vector_size)

    vocabulary_len = len(vocabulary)
```

[1]Source: https://github.com/RaRe-Technologies/gensim/blob/05ca318eebf934cd87c019d94bf4fab25ead802a/gensim/models/keyedvectors.py#L917

```python

    distances = np.zeros((vocabulary_len, vocabulary_len))
    distances[np.ix_(doc1_idx, doc2_idx)] = euclid(
        doc1_embeddings,
        doc2_embeddings
    ).reshape(
        len_tokens_doc1,
        len_tokens_doc2
    )

    # Get normalized BOW reprsentations of the documents as dense
    arrays
    bow_doc1 = np.zeros(vocabulary_len)
    bow_doc2 = np.zeros(vocabulary_len)

    data_doc1, ind_col_doc1, _ = bag_of_words([doc1], vocabulary,
    normalize=True)
    data_doc2, ind_col_doc2, _ = bag_of_words([doc2], vocabulary,
    normalize=True)

    bow_doc1[ind_col_doc1] = data_doc1
    bow_doc2[ind_col_doc2] = data_doc2

    return emd(bow_doc1, bow_doc2, distances)


def word_movers_distance(X_q1, X_q2, model):
    return np.array([
        word_movers_distance_document_pair(doc1, doc2, model)
        for doc1, doc2 in zip(X_q1, X_q2)
    ])
```

3.2.2 Irene's features

I have implemented a feature vector and a simple distance metric. The feature vector consists on applying singular value decomposition (SVD) to a co-occurrance matrix to generate word representations. A co-occurance matrix is a $n \times n$ matrix where each row contains the embedding of a word i, which is the number of times the word co-occurrs in a sentence with all the other words in the vocabulary. The idea behind this is that two words that have a very similar meaning, will often be found in the same context, co-occuring with the same words (e.g. the fruits orange and apple are both likely to be found in the sentence "I drank a «fruit» juice", together with the words "drank" and "juice"). Therefore, a word is defined by the context where it is used. As this generates a huge sparse matrix, we then apply SVD to capture the main axis of variation of the dataset in a lower dimensional space. As done with the word2vec embeddings, we then represent a sentence as the weighted sum of the embedding of the words it contains. The word embeddings are scaled by multiplying them by the word's TF-IDF values, giving more importance to document-specific words. For comparability with the selected word2vec embeddings, we have selected the top 300 components to represent the word.

Regarding the implementation, I have used the scipy dok sparse matrix type to compute the co-occurrance matrix, which allows to iteratively update the values of the matrix while storing it on an efficient way. Computing the matrix is as easy as iterating over all the questions in our vocabulary and summing 1 to each pair of words in the sentence. For SVD, I have used the scipy svds function, which operates with sparse matrices. I have also included two functions to visualize the representation of each sentence using the two first components, which allows to explore how well the embeddings are performed. I have observed that the first components capture only a small amount of variation, meaning that the matrix is not very compressible and that we are losing a lot of information. This explains the poor performance obtained with the models based on these embeddings.

I have also implemented the Jaccard similarity metric, which computes the similarity between two sentences as the ratio of the number of common words divided by the total number of unique words. The intersection and the union can be easily computed using python sets.

3.2.3 Otis' features

The similarity between the sentences can directly be computed using metric measures on the embeddings computed using tf-idf and WMD presented in Vladislav's section. Two logical candidates are the Euclidean and Manhattan distance, or $\|\mathbf{A}-\mathbf{B}\|_2$ and $\|\mathbf{A}-\mathbf{B}\|_1$ respectively.

An alternative measures is the cosine similarity, which is a popular measure in the domain of NLP. This uses the property of the embedding space as an inner product space and combines the angle between two vectors. As a result, the magnitude is irrelevant. The cosine similarity is written as

$$\cos(\theta) = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|},$$

where the right side of the equation expresses its computation. As the cosine similarity is a similarity measure and not a distance, larger values signify a lower distance. We term cosine distance 1 - cosine similarity. The range is from 0 to 1.

The corresponding functions can be found in the `utils.py` file.

4 Final results (Otis)

We have chosen to train a logistic regression and an XGBoost classifier with two final models and a baseline. We have decided to use a logistic regression for comparability with the baseline and XGBoost because it is a state of the art classifier that can be easily trained. Our baseline feature vector consists on the bag of words representation of the questions. For our improved solution we have designed two different feature vectors, relying each of them in a different word embedder (co-occurrance-SVD and word2vec). For the two of them, we have computed sentence embeddings from a weighted sum of its word vectors, where the weight is the TF-IDF value for each word. Using these sentence embeddings, we have computed the distance between the questions with four differences distance metrics, and also its jaccard similarity by directly comparing the words. To compute distances between embeddings we have used cosine, Euclidean and Manhattan distance, in addition to WMD. The two sentence embeddings and the 5 distance metrics are concatenated and passed to the logistic and XGBoost classifiers.

As the distance metrics are a direct representation of the similarity between sentences, we can use them as the sole predictor of equivalence and plot their ROC curve and compute the AUC. The results can be found in figure 1. Unsurprisingly, we can see that word2vec embeddings perform better than co-occurrance embeddings, as they are more advanced embeddings, trained on a large corpus and able to capture contextual information. As mentioned, our co-occurrance matrix does not allow much compression in terms of variance explained by the first components, which means that a lot of information is lost. Regarding the distance metrics, we can see that the Jaccard index offers a poor performance, which is unsurprising considering that it is a metric that works directly on the sentence, without incorporating any knowledge derived from corpus on words with similar meanings and synonyms. For word2vec the cosine similarity is the weakest measure, which is surprising considering its wide usage in similar embeddings compared to the other measures. The other measures perform similarly, with the highest AUC of 0.7401 produced by Euclidean distance. The WMD performs better than Jaccard distance but worse than the other metrics. In the case of co-occurrance-SVD, we can see that only cosine distance offers a performance better than Jaccard, suggesting that the size of the vectors expresses features irrelevant to the task at hand. As cosine distance is insensitive to vector size, the measure better captures meaningful differences between embeddings. As the data was not scaled before applying the SVD, the values express absolute as opposed to relative quantities meaning that longer sentences have higher values and the euclidean and manhattan distances are incompatible. This could be resolved by normalizing the data.

As can be seen in figure 2 the model with the best performance is the XGBoost with word2vec embeddings (AUC=0.8765 in validation set), in comparison to a baseline of AUC=0.8096 on logistic regression with BoW embeddings.This is expectable as the word2vec embeddings are more expressive and XGBoost a more powerful classifier. The co-occurrance-SVD embeddings with XGBoost also outperforms the baseline (AUC=0.8407). Interestingly, logistic regression performs worse for either of the two proposed improved embeddings. This means that the BoW embeddings are more linearly separable.

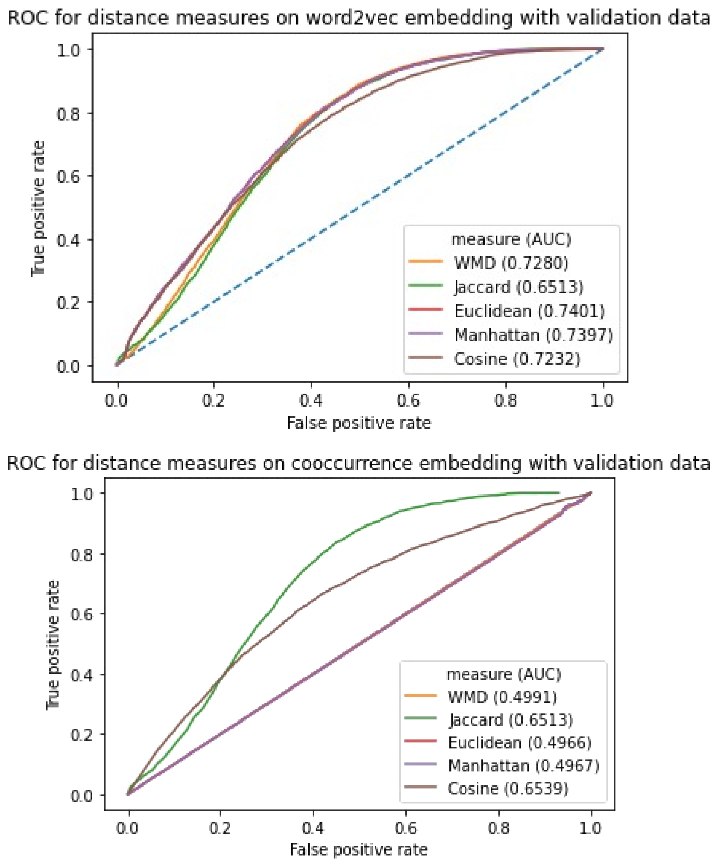

Figure 1: ROC curve for the various distance measures between the questions as predictors on the validation set. The AUC is given between parentheses

5 Conclusions

By using an embedding that better expresses the relationship between words and applying a more complex model, we have attained a considerable improvement over the baseline model. In particular, a TF-IDF weighted sum of word2vec embeddings, combined with several distance metrics and an XGBoost classifier has offered the best performance. We expect that using more sophisticated techniques, for instance recurrent neural networks or transformers, which allow for deeper expression of relationships, would bring even higher improvements.

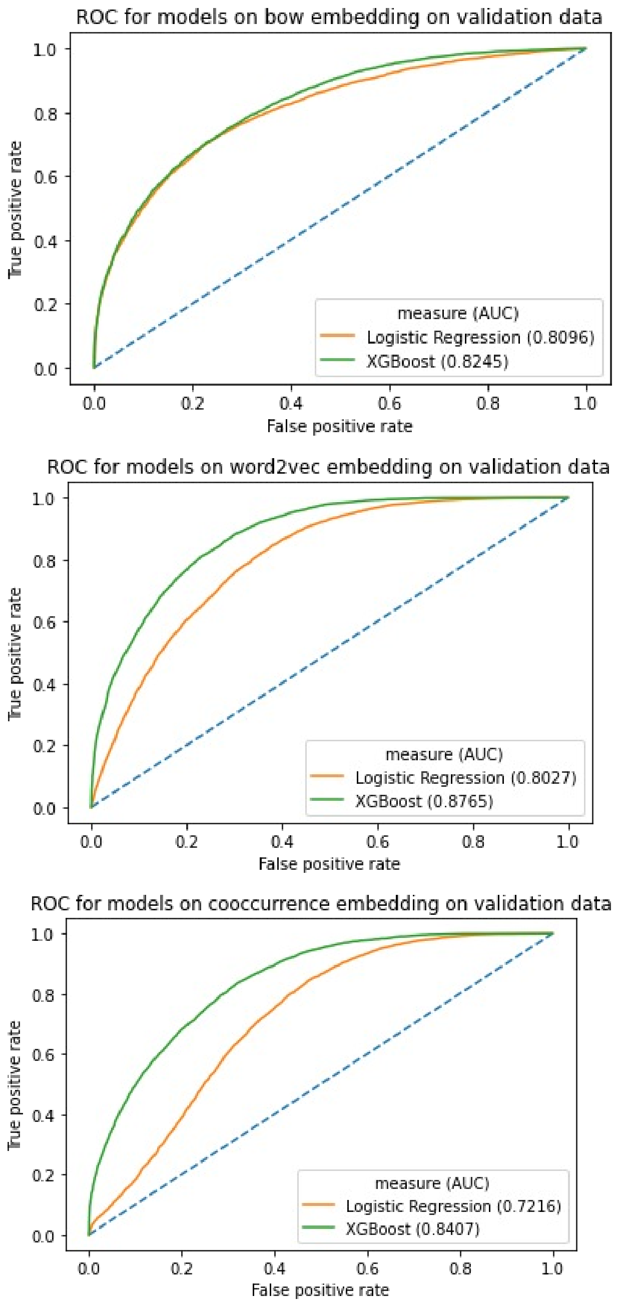

Figure 2: ROC curve for the models combining the different embeddings and distance measures between the questions on the validation set. The AUC is given between parentheses

References

[1] M. Kusner, Y. Sun, N. Kolkin, and K. Weinberger. From word embeddings to document distances. In F. Bach and D. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 957–966, Lille, France, 07–09 Jul 2015. PMLR. URL `https://proceedings.mlr.press/v37/kusnerb15.html`.