UNIVERSITAT DE BARCELONA

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

Explain what we did (if we ended up doing something; if not, remove this section and any reference from the previous paragraph)

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 Irene's features

3.2.2 Otis' features

3.2.3 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


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

```
68
69          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:

```
1  def bag_of_words(documents, vocabulary, normalize=False):
2      data = []
3      ind_col = []
4      ind_ptr = [0]
5
6      for doc in documents:
7          bow_doc = defaultdict(int)
8
9          for word in doc:
10             if word in vocabulary:
11                 bow_doc[word] += 1
12
13         bow_array = np.array(list(bow_doc.values()))
14         bow_norm = np.sum(bow_array) if normalize else 1.0
15
16         bow_doc_normalized = [
17             bow_doc[word] / bow_norm
18             for word in bow_doc.keys()
19         ]
20
21         cols = [vocabulary[word] for word in bow_doc.keys()]
22
23         data.extend(bow_doc_normalized)
24         ind_col.extend(cols)
25         ind_ptr.append(len(ind_col))
26
27     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
        ]

        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:

```
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)

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

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

```
        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)
    ])
```

4 Final results

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`.