UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Embeddings and Supervised Machine Learning Models

LNR. Degree in Data Science

May 2024

Javier Luque & Martin Ruddy

Index

List of Figures

1 Introduction

The main objective is to train and validate several supervised classification models for addressing the tasks proposed in the shared task "Oppositional thinking analysis: Conspiracy theories vs critical thinking narratives".[1]

In order to train the various models, we will provide as input different text representations: traditional forms, static embedding-based and contextual embedding-based). Of course, the text will need to be processes appropriately before it gets its representations. Once we have our data preprocessed and split into 90% for train and 10% for test, we can carry out with our project.

Firstly, we will train supervised machine learning methods such as Support Vector Machines (SVMs), Logistic Regression, Decision Trees, and Multilayer Perceptrons (MLPs). Secondly, we will combine some of these machine learning models with ensemble techniques, including Voting, Stacking, Bagging, Boosting and Random Forests (RF). By harnessing the collective strength of the simple models, ensembles can achieve better performance than using isolated models.

Finally, the models' performance will be evaluated with the Matthews correlation coefficient.

2 Data Overview

The official training dataset for both languages are in the file Dataset-Oppositional.zip. This file provides two json files with the training data (although it will be split itself into 90/10), one for each language. This json files have an independent variable, which is a text; and a dependent variable, which is either 'CONSPIRACY' or 'CRITICAL', indicating the real classification of the text. The goal for the models trained will consist on predicting if a text is either 'CONSPIRACY' or 'CRITICAL'.

```python
import json

def load_dataset_from_json(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        data = json.load(file)
    return data

dataset = load_dataset_from_json('dataset_en_train.json')
dataset_spanish = load_dataset_from_json('dataset_es_train.json')
```

Figure 1: Data loading

[1] For an extended explanation of this task, consult [1].

3 Data Preproccess

As a first step in the project, we have split the provided data for the challenge into separate train and test datasets, through the use of sklearn's train_test_split function. The size of the train dataset is 90% of all text samples, while the size of the test dataset is the remaining 10%. The original dataset does not contain missing values, so regarding the data preprocessing task, not a lot of work was needed.

In order to be able to use the data for machine learning classification algorithms, it is necessary to embed all the provided texts into numeric vectors, so they can be utilised by the models. This step is a crucial one, since depending on the method that we choose to embed the texts, a different result might be produced at a later stage in the analysis process.

However, before we embed our texts, we will need to make some small modifications, which are accomplished by the function in preprocess() function

```python
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
import string

nltk.download('stopwords')
nltk.download('punkt')

def preprocess(text_list, english=True):
    answer = []
    if english:
        stop_words = set(stopwords.words('english'))
    else:
        stop_words = set(stopwords.words('spanish'))
    punctuation = set(string.punctuation)
    for text in text_list:
        tokens = word_tokenize(text.lower())
        tokens = [token for token in tokens if token.isalnum() and token not in stop_words and token not in punctuation]
        answer.append(' '.join(tokens))
    return answer
```

Figure 2: preprocess() function

In essence, this function does two things: it converts all the text to lowercase and it removes the stop words (different for each language) and the punctuation marks.[2] [3]

Now we can move on with the embedding process. Text embedding can be achieved through the use of different techniques, and each technique can be applied through different models. The embedding techniques that we have considered for this project are the following:

- Traditional forms.

[2] Keep in mind that the function receives a list of texts, as each text belongs to a single observation.
[3] Some embedding techniques (particularly Contextual Word Embeddings) do not require this preprocessing.

- Static Word Embeddings.
- Contextual Word Embeddings.

For each of these we have considered several embedding models, with the aim of finding which one can achieve the best embeddings with regard to the classification task at the analysis step.

3.1 Traditional forms

3.1.1 Bag of Words (BoW)

BoW is an easy and effective way of transform text into a numerical representation. In this representation, each text in the dataset is tokenized. Then, a vocabulary is obtained from all tokens (lowercase words) in the entire dataset. A vector is obtained for each text. Each axis represents the number of times that a token (in the vocaboulary) appears in the text.

The corpus is represented by a matrix with one row per text instance and column per token in the vocabulary. It is a high-dimensional and sparse representation (each term in the vocabulary represents a feature). The order of the words in the text is ignored.

To embed our dataset with Bag of Words we used the code of Bag of Words Embedding. Please note that **'datas'** is a list that contains the following datasets (in order): train english dataset, test english dataset, train spanish dataset and test spanish dataset. We will refer this variable as this list of datasets in the rest of the document.

```python
from sklearn.feature_extraction.text import CountVectorizer

### BAG OF WORDS

def create_vecs(vectorizer):
    vecs = []
    i = 5
    for X in datas:
        if i%2 == 0: # The odds are train, the even test
            vecs.append(vectorizer.transform(preprocess(X)))
        else:
            vecs.append(vectorizer.fit_transform(preprocess(X)))
        i += 1
    return vecs

vectorizer_bw = CountVectorizer()
vectorizers["Bag of Words"] = create_vecs(vectorizer_bw)
```

Figure 3: Bag of Words Embedding

3.1.2 Binary Bag of Words

It is the same as the previous one with only one slight difference, it is simpler. As its names indicates, it has binary values: 1 if the feature occurs at least one, and 0 otherwise.

```python
from sklearn.feature_extraction.text import CountVectorizer

### BINARY BAG OF WORD

def create_vecs(vectorizer):
    vecs = []
    i = 5
    for X in datas:
        if i%2 == 0: # The odds are train, the even test
            vecs.append(vectorizer.transform(preprocess(X)))
        else:
            vecs.append(vectorizer.fit_transform(preprocess(X)))
        i += 1
    return vecs

vectorizer_bbw = CountVectorizer(binary=True)
vectorizers["Binary Bag of Words"] = create_vecs(vectorizer_bbw)
```

Figure 4: Binary Bag of Words Embedding

3.2 Static Word Embeddings

Static embeddings assign a fixed vector representation to each word in the vocabulary, regardless of its context within a sentence or document. Techniques like Word2Vec, GloVe, and FastText are grouped in this category. While static embeddings capture general semantic meanings of words, they fail to capture nuances that arise from varying contexts.

3.2.1 Word2Vec

Word2Vec is a popular algorithm used to generate word embeddings, which are dense vector representations of words in a continuous vector space. It consists of two main models: Continuous Bag of Words (CBOW) and Skip-gram.

- *Continuous Bag of Words (CBOW).* CBOW predicts the target word based on the context words surrounding it. It takes a context window of surrounding words and tries to predict the target word in the middle. The model learns to predict the target word by minimizing the difference between the predicted probability distribution and the actual distribution of words.
- *Skip-gram.* Skip-gram, however, predicts the context words given a target word. It takes a target word and tries to predict the context words that will likely appear around it. The model learns to predict the context words by maximizing the probability of observing them given the target word.

Both CBOW and Skip-gram are trained using a large corpus of text data and learn to generate word embedding vectors that capture semantic relationships between words based on their co-occurrence patterns in the text.

If you want to use Word2Vec with texts in english, you just have to load the 'word2vec-google-news-300' pre-trained model from the gensim API, as shown in Word2Vec Embedding. However, for spanish text is a little bit trickier, you will have to load the pre-trained embedding matrix from a local file with the KeyedVectors class, as we did. Many of these files can be found online in several languages, we got ours from `http://vectors.nlpl.eu/repository/`

```python
import gensim.downloader as api
from gensim.models.keyedvectors import KeyedVectors

### WORD2VEC

def get_sentence_rep(text, model, dim=300):
    words = text.lower().split()
    zero_vec = np.zeros(dim)
    avg_vec = np.zeros(dim)
    total_w = 0
    for w in words:
        try:
            avg_vec += model.get_vector(w)
            total_w += 1
        except:
            pass
    if total_w == 0:
        return zero_vec
    return avg_vec/total_w

vecs = []
model_word2vec_en = api.load('word2vec-google-news-300')
print("English model loaded")
vecs.append([get_sentence_rep(text, model_word2vec_en) for text in preprocess(datas[0])])
vecs.append([get_sentence_rep(text, model_word2vec_en) for text in preprocess(datas[1])])
print()
model_word2vec_es = KeyedVectors.load_word2vec_format('pre-trained-spanish/68/model.bin', binary=True, limit=None)
print("Spanish model loaded")
vecs.append([get_sentence_rep(text, model_word2vec_es, dim=100) for text in preprocess(datas[2])])
vecs.append([get_sentence_rep(text, model_word2vec_es, dim=100) for text in preprocess(datas[3])])
print()
vectorizers["Word2Vec"] = vecs
```

Figure 5: Word2Vec Embedding

3.3 Contextual Word Embeddings

Contextual embeddings, unlike static embeddings, consider the surrounding context of each word to generate its embedding. Models such as ELMo and BERT are prominent examples. By leveraging deep learning architectures like recurrent neural networks (RNNs) or transformers, contextual embeddings capture the dynamic nature of language, providing richer representations that adapt to the context of each word within a sentence.

3.3.1 BERT

BERT is described as one of the most significant breakthroughs in the history of (Google's) Search. Specifically, it is used in the disambiguation of queries in all searches in English, but other languages are also included, such as Spanish, Portuguese, Hindi, Arabic, German, etc. BERT has also been used in many automatic text classification tasks and other Natural Language Processing tasks.

The general idea of building a BERT-based classifier of two essential stages, as seen in Figure 6. The first is to obtain the (deep) representation of the texts from the input data and use said representation as input to a classifier,

which is usually an MLP that recieves the output of the BERT model as input and outputs a layer with as many neurons as classes in the task.

It is essential to highlight that the BERT model can be used directly from general data with which it was trained (pre-trained), or it can be fine-tuned to consider specific aspects of the task to be solved (fine-tuned).

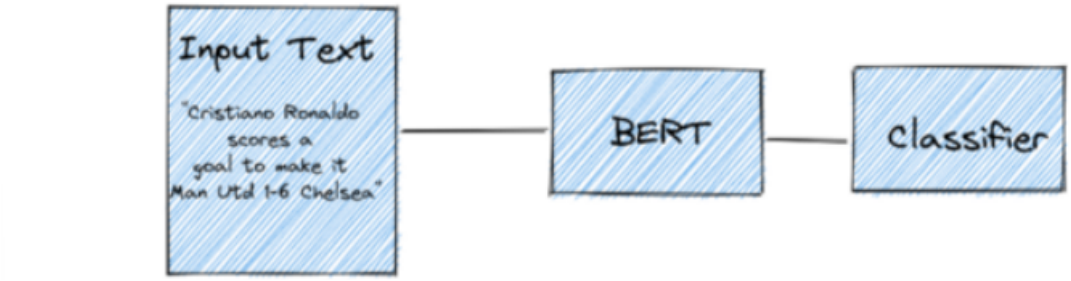

Figure 6: A general architecture for using BERT in classification tasks

The BERT architecture consists of stacked encoders. Each encoder encapsulates two sublayers: a self-attention layer and a feed-forward layer.

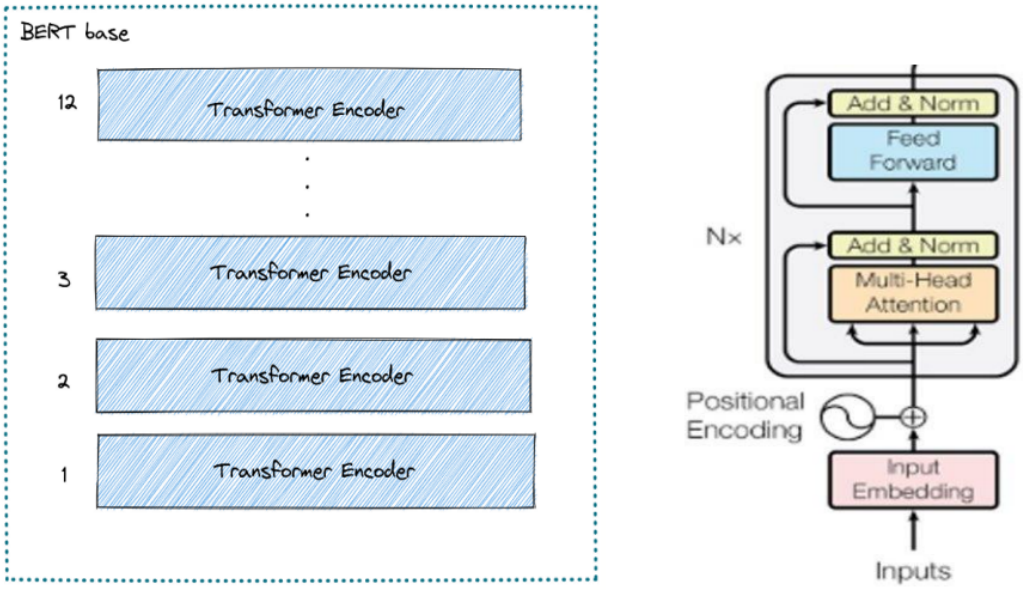

Figure 7: BERT Architecture and Transformer Encoder

There are different models of BERT regarding the size of the architecture:

- Base BERT is a model consisting of 12 encoder layers, 12 self-attention heads, 768 hidden sizes, and 110 million parameters.
- Large BERT is a model consisting of 24 encoder layers, 16 self-attention heads, 1024 hidden sizes, and 340 million parameters.

BERT is a robust language model because it is pre-trained with unlabeled data from BooksCorpus (800 million words) and Wikipedia (2.5 billion words). Plus, it is pre-trained using the bidirectional nature of stacked encoders. This fact means that BERT learns the information in a sequence of words from left to right and from right lo left.

The BERT model requires a sequence of tokens (words) as its input. Within each token sequence, there are tow special tokens that must be included according to BERT's expectations, as shown in Figure 8:

- [CLS] This token marks the beginning of each sequence and is known as the classification token.
- [SEP] This token serves the purpose of informing BERT about the boundaries between sentences within the token sequence. It plays a crucial role, especially in tasks involving next sentence prediction or question answering. In cases where there is only one sentence, this token is added at the end of the sequence.

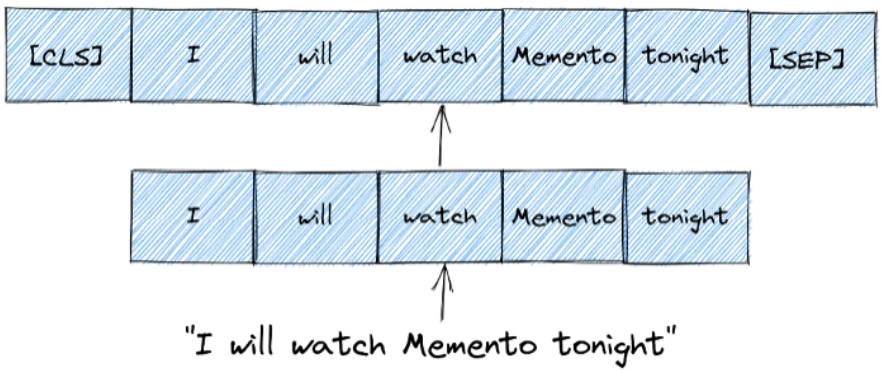

Figure 8: Tokenized textual input

Transforming the input sentence into a token sequence for BERT requires just a single line of code. To achieve this, we utilize the BertTokenizer class, which will be demonstrated later. It is crucial to consider that the maximum allowable token length for the BERT model is 512. If a sequence's tokens fall short of this limit, sequence completion with the [PAD] token can be applied. On the other hand, if the token sequence surpasses the 512-token limit, truncation is performed accordingly.

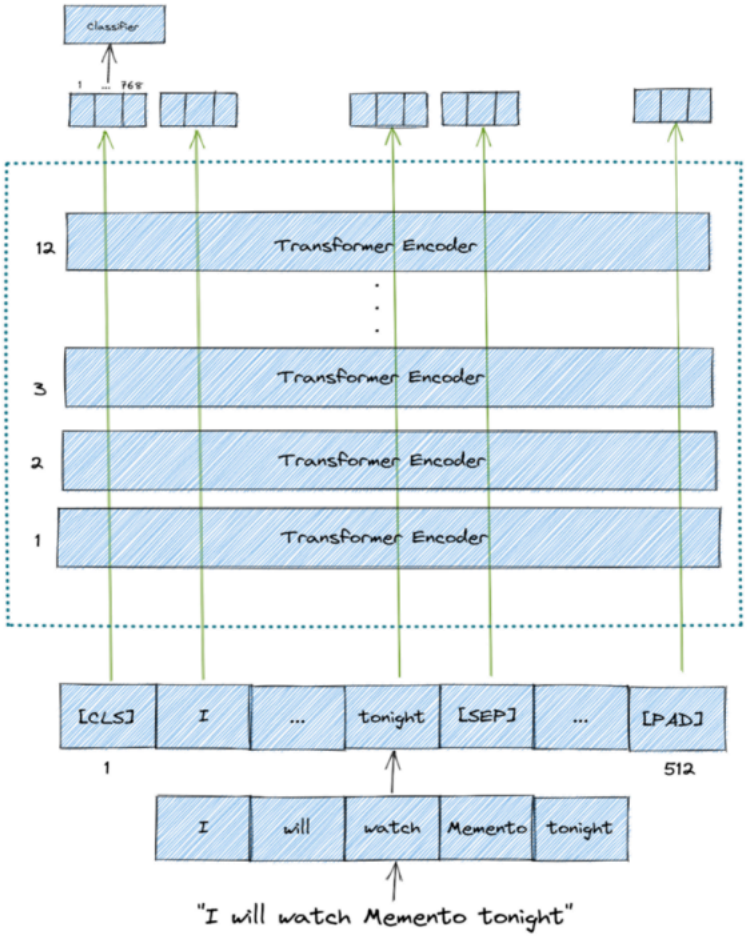

Figure 9: BERT output for a textual input

The BERT model then generates a 768-dimensional embedding vector for each token. These vectors can serve as input for various Natural Language Processing applicatoins, including text classification, next sentence prediction, Named Entity Recognition (NER), and Question & Answering.

In the case of text classification, we focus on the embedding vector output of the special token [CLS]. This entails utilizing the 768-dimensional embedding vector of the [CLS] token as input for the classifier (MLP). The classifier, in turn, generates a vector of a size corresponding to the number of classes in the classification task. This process is illustrated in Figure 9.

As we discussed we will use the BertTokenizer and BertModel classes to load the necessary models for our classification task. These pretrained model will be obtained from: 'bert-base-uncased' for english and 'dccuchile/bert-base-spanish-wwm-uncased' for spanish.

```python
import torch
from transformers import BertTokenizer, BertModel

### BERT

def get_sentence_rep2(text, model, tokenizer):
    inputs = tokenizer(text, padding=True, truncation=True, return_tensors="pt")
    with torch.no_grad():
        outputs = model(**inputs)
    cls_vector = outputs[0][:, 0, :]  # Get the CLS vector
    return cls_vector

token_bert = BertTokenizer.from_pretrained('bert-base-uncased')
model_bert = BertModel.from_pretrained('bert-base-uncased')
token_bert_es = BertTokenizer.from_pretrained('dccuchile/bert-base-spanish-wwm-uncased')
model_bert_es = BertModel.from_pretrained('dccuchile/bert-base-spanish-wwm-uncased')

vecs = []
print("English vectorization")
vecs.append([get_sentence_rep2(text, model_bert, token_bert) for text in datas[0]])
vecs.append([get_sentence_rep2(text, model_bert, token_bert) for text in datas[1]])
print("COMPLETED")

print("Spanish vectorization")
vecs.append([get_sentence_rep2(text, model_bert_es, token_bert_es) for text in datas[2]])
vecs.append([get_sentence_rep2(text, model_bert_es, token_bert_es) for text in datas[3]])
print("COMPLETED")
vectorizers["BERT"] = vecs
```

Figure 10: BERT Embedding

3.3.2 RoBERTa

Robustly Optimized BERT Pre-training Approach is a variant of the popular BERT (Bidirectional Encoder Representations from Transformers) language model, introduced by researchers from Facebook AI Research in 2019. RoBERTa was trained with a larger batch size of 8000 samples, compared to 256 for BERT. This allowed for better utilization of GPU memory and faster training.

RoBERTa was trained for longer, with more computational resources, resulting in improved performance. conversely to BERT, which was trained on both Masked Language Modeling (MLM) and Next Sentence Prediction (NSP) tasks. RoBERTa removed the NSP task and focused solely on the MLM task, which proved to be more effective.

Moreover, the model used a larger BPE vocabulary of 50000 tokens, compared to 30000 for BERT, allowing for better representation of rare words. Instead of using the same masking pattern for all examples, RoBERTa used a different masking pattern for each example during training, which

improved generalization.

These modifications, along with other minor changes, resulted in RoBERTa achieving significantly better performance than BERT onn a wide range of Natural Language Processing tasks, such as question answering, text classification, and language inference, while using the same architecture as BERT.

Exactly as we did with BERT, we will obtain the pre-trained models from (english) and (spanish) with RobertaTokenizer and RobertaModel.

```
import torch
from transformers import RobertaTokenizer, RobertaModel

### ROBERTA

token_roberta = RobertaTokenizer.from_pretrained('roberta-base')
model_roberta = RobertaModel.from_pretrained('roberta-base')
token_roberta_es = RobertaTokenizer.from_pretrained('PlanTL-GOB-ES/roberta-base-bne')
model_roberta_es = RobertaModel.from_pretrained('PlanTL-GOB-ES/roberta-base-bne')

vecs = []
print("English vectorization")
vecs.append([get_sentence_rep2(text, model_roberta, token_roberta) for text in datas[0]])
vecs.append([get_sentence_rep2(text, model_roberta, token_roberta) for text in datas[1]])
print("COMPLETED")

print("Spanish vectorization")
vecs.append([get_sentence_rep2(text, model_roberta_es, token_roberta_es) for text in datas[2]])
vecs.append([get_sentence_rep2(text, model_roberta_es, token_roberta_es) for text in datas[3]])
print("COMPLETED")
vectorizers["RoBERTa"] = vecs
```

Figure 11: RoBERTa Embedding

3.4 Other Considerations

Besides these embeddings, we have tried many more traditional ones, which can be seen in vectorization.py. This saves a pickle file with the embeddings, file which be later be loaded to train the models. This also saves in other pickle file the dependet variable (english train, english test, spanish test and spanish train).

Once this phase is completed, and the separated datasets have been embedded with the different selected techniques, we were ready to move to the next step: training machine learning models on the embedded datasets to accurately predict the type of text each is referring to.

4 Models' Training

4.1 Simple Models

4.1.1

4.2 Ensembles

5 Model's Evaluation

5.1 SVC

6 Overview / Conclusiones (eln ombre que sea)

7 Bibliography

[1] PAN. Oppositional thinking analysis: Conspiracy theories vs critical thinking narratives. Damir Korenčić, Berta Chulvi, Xavier Bonet, Mariona Taulé, Paolo Rosso, Francisco Rangel https://pan.webis.de/clef24/pan24-web/oppositional-thinking-analysis.html

[2] Wikipedia. Phi Coefficient https://en.wikipedia.org/wiki/Phi_coefficient

[3] GitHub. pan-clef-2024-oppositional. Damir Korenčić https://github.com/dkorenci/pan-clef-2024-oppositional/tree/main

[4] LNR, Grado en Ciencia de Datos, UPV. Preprocessing and Traditional Language Representation using scikit-learn. Reynier Ortega https://poliformat.upv.es/access/content/group/GRA_14029_2023/Lab/Lab-PL2S1/PW2_S1.pdf

[5] LNR, Grado en Ciencia de Datos, UPV. Static and Contextual word-embeddings for text representation. Reynier Ortega https://poliformat.upv.es/access/content/group/GRA_14029_2023/Lab/Lab-PL2S2/PW2_S2.pdf

[6] NLPL word embeddings repository http://vectors.nlpl.eu/repository/

[7] LNR, Grado en Ciencia de Datos, UPV. ontextual Word Embeddings: ELMo and BERT-based models. Reynier Ortega https://poliformat.upv.es/access/content/group/GRA_14029_2023/Lab/Lab-PL2S3/PW2_S3.pdf