Module_2_Report

April 4, 2022

1 Fake News Detection - Module 2

#Agenda

1. Project Description & Objectives Recap
2. Models Description & Analysis
3. Conclusion
4. Appendix

2 Project Description and Objectives

Early fake news detection aims to give early alerts of fake news during the dissemination process before it reaches a broad audience. Therefore, early fake news detection methods are highly desirable and socially beneficial.

- **What is the goal of your models and what are you trying to achieve?**
  The goal of this project is to detect fake news shared on social media, specifically on Twitter. A binary classification machine learning model will be developed to detect patterns in the article or post, it will classify the news as Fake(1) or Real(0).

  The texts that we have are the title and the content of the news article. The numerical information that we have are information about the article: the source transformed into categorical variables and the date of publication. We also have information extracted from Twitter about the article, about the tweets reacting to each article: number of tweets, date of publication, the number of likes, retweets and replies.

  Our goal is to use models to process the text, the title and the content of the article, and models to process the numerical features. Then, we would like to combine both models to take advantage of both performance to have a final outcome about the article: if it is fake or real news.

- **What models and methodologies have you used?**
  There are two stages in the modeling part:
  - First, we explored different NLP techniques to process the text and transformed into texture features. We also tried to feed the texture features in different binary classification models. This is to determine which NLP techniques have the best performance.
  - Then, we concatenated both texture and non-texture features. Again, we fed all features into binary classfication models for final comparison.

To compare the performance of our models, we looked at the following performance metrics: f1 score (weighted average of Precision and Recall), Accuracy, TPR and FPR. We chose to attribute the same weight to both type of errors: False Positive and False Negative. According to us, both errors could have bad consequences, because our aim is to spot every fake news so to prevent False Positive errors, but we also wanted to keep the number of False Negative errors as low as possible, because if a real news is labelled as fake and is not published because of that, it would be censorship. To avoid the errors, we chose to fit the hyperparameters of our models (when necessary) by maximizing the f1 score. The f1 score is defined as the following:

3 Models Description and Texture Features Analysis

3.0.1 Bag of words

- **How do these models work?**

  The bag of words model is a very intuitive model of Natural Language Processing. The goal is to select to most frequent words of the text and add them as catgeorical features to our models (if the word is present in the text the feature is equal to 1 and otherwise to 0).

  We first did some preprocessing on the text (for the training, testing and validation sets) : we removed digits, punctuation, tokenized the text, removed stop words and stemmed words and finally detokenized the text.

  We then used **Count Vectorizer** from *'sklearn.feature_extraction.text'* to select the most frequent words from the training set for a given frequency. We chose the frequency in order to make our bag-of-words model faster to run but have enough information to improve the model performance. We selected a frequency of 2% for the titles, and 30% for the bodies : the difference is explained by the fact that there are less words in the articles' titles than bodies.

  Once we selected the words from title and body we added them as features while adding '_title' or '_body' to differentiate words from the body and from the title.

  Finally we run several classification models on the pre-processed data such as: a baseline model (predicting the most frequent outcome i.e. no fake news), a Logisitic Regression with a threshold at 0.5, a Classification Tree, a Random Forest and finally a XGBoost model.

- **Why have you chosen these models?**

  We chose the bag-of-words model because it has a good interpretability and it was a good model to start with and understand what performance to expect on other models.

  We have chosen to combine our bag-of-words with a Logisitic Regression with a threshold at 0.5, a Classification Tree, a Random Forest and finally a XGBoost model, because these are common classification models. We first tried relatively simple models as a Logistic Regression and a Classification Tree because we knew they would be shorter to run in comparison to the Random Forest and XGBoost (which are known for having a better performance in general).

- **How have you selected any tuning parameters for your models (a.k.a hyperparameters)?** For the CART and the Random Forest, we did some cross-validation with GridSearchCV, in order to maximize the f1 score (that we selected as our main performance

metric). For the Logistic Regression, we chose a threshold of 0.5 as we decided that the classes 'fake news' and 'reals news' have the same weight. For the XGBoost model, we did some parameters tuning on the validation set without GridSearchCV. Indeed, the model was to long to run with GridSearchCV.

- **How well does your model perform and how have you measured that?** We measured the performance by running our model on the testing set considering several metrics : f1-score, accuracy, TPR and FPR. However, we put our focus especially on the f1-score that we considered as the most important. The performance metrics are given by the table below. The model does not perform extremely well: the f1-score is between 0.413 and 0.449, the baseline for accuracy is 0.716 and our bag-of-words models have an accuracy between 0.742 and 0.765. The model performing the best is the Random Forest, performing better than XGBoost probably because of a more precise parameters tuning (use of GridSearchCV).
- **Are there any significant limitations to your modeling methods (i.e. Where, when, how, and why can your model fail?)** The main limitation of the bag-of-words model is that it takes a long time to run, which would make it difficult to train on more data and difficult to scale it.

[2]:
```
#@title
import pandas as pd
comparison_data = {'Baseline model': ['{:.3f}'.format(0.000),
                                      '{:.4f}'.format(0.716),
                                      '{:.3f}'.format(0.000),
                                     '{:.4f}'.format(0.000)],
                   'Logistic Regression': ['{:.3f}'.format(0.419),
                                      '{:.4f}'.format(0.759),
                                      '{:.3f}'.format(0.305),
                                      '{:.4f}'.format(0.060)],
                   'Classification Tree': ['{:.3f}'.format(0.413),
                                           '{:.4f}'.format(0.742),
                                           '{:.3f}'.format(0.319),
                                            '{:.4f}'.format(0.090)],
                   'Random Forest': ['{:.3f}'.format(0.449),
                                     '{:.4f}'.format(0.765),
                                     '{:.3f}'.format(0.336),
                                     '{:.4f}'.format(0.064)],
                   'Gradient Boosting': ['{:.3f}'.format(0.440),
                                         '{:.4f}'.format(0.764),
                                         '{:.3f}'.format(0.326),
                                         '{:.4f}'.format(0.062)]}

comparison_table = pd.DataFrame(data=comparison_data,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[2]:

	Baseline model	Logistic Regression	Classification Tree	Random Forest	\
f1-score	0.000	0.419	0.413	0.449	
accuracy	0.7160	0.7590	0.7420	0.7650	

```
TPR              0.000              0.305              0.319          0.336
FPR             0.0000             0.0600             0.0900         0.0640

          Gradient Boosting
f1-score             0.440
accuracy            0.7640
TPR                  0.326
FPR                 0.0620
```

Performance after adding the non-text features - We tried several models by adding the non-text related features to the columns of our dataset and ran the different models. - We got the following performance as a result:

[3]:
```
#@title
comparison_data = {'Baseline model': ['{:.3f}'.format(0.000),
                                      '{:.3f}'.format(0.716),
                                      '{:.3f}'.format(0.000),
                                     '{:.3f}'.format(0.000)],
                   'Logistic Regression': ['{:.3f}'.format(0.469),
                                      '{:.3f}'.format(0.773),
                                      '{:.3f}'.format(0.352),
                                      '{:.3f}'.format(0.059)],
                   'Classification Tree': ['{:.3f}'.format(0.716),
                                           '{:.3f}'.format(0.857),
                                           '{:.3f}'.format(0.634),
                                            '{:.3f}'.format(0.054)],
                   'Random Forest': ['{:.3f}'.format(0.769),
                                     '{:.3f}'.format(0.880),
                                     '{:.3f}'.format(0.704),
                                     '{:.3f}'.format(0.051)],
                   'Gradient Boosting': ['{:.3f}'.format(0.779),
                               '{:.3f}'.format(0.884),
                               '{:.3f}'.format(0.715),
                               '{:.3f}'.format(0.048)]}

comparison_table = pd.DataFrame(data=comparison_data,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[3]:
```
          Baseline model Logistic Regression Classification Tree Random Forest  \
f1-score           0.000               0.469               0.716         0.769
accuracy           0.716               0.773               0.857         0.880
TPR                0.000               0.352               0.634         0.704
FPR                0.000               0.059               0.054         0.051

          Gradient Boosting
f1-score              0.779
```

accuracy	0.884
TPR	0.715
FPR	0.048

The bag-of-words model is performing significantly better on the model including non-text features, especially for the Classification Tree, the Random Forest and the Gradient Boosting models.

The hyperparameters for these models were selected by cross-validation on the f1-score with GridSearchCV, except for XGBoost for which we used the valisation set for our cross validation.

3.0.2 TF-IDF

- **How does this model work?**

  The TF-IDF (term frequency-inverse document frequency) model is a statistical measure that evaluates how relevant a word is to a document in a collection of documents. This model will consider each word of the articles as a feature, and attribute a numerical vector to this word to assess the relevance of this word to determine the outcome.

  This requires preprocessing on the text on the three data sets that we have (training, test and validation). Similarly to the preprocessing for bag of words, we removed digits, punctuation, tokenized the text, remove stop words and lemmatize each word (i.e. transform them into a basic form, so that a verb conjugates in different ways refer to a same word for example).

  We then used **TfidfVectorizer** from *'sklearn.feature_extraction.text'* to compute the numerical vector associated to each word. This numerical vector is computed by multiplying two metrics: how many times a word appears in a document (term frequency or TF), and the inverse document frequency of the word across a set of documents. This means, how common or rare a word is in the entire document set. The closer it is to 0, the more common a word is. This metric can be calculated by taking the total number of documents, dividing it by the number of documents that contain a word, and calculating the logarithm.
  Mathematically, the TF-IDF score for the word t in the document (article) d from the document (article) set D is calculated as follows:

Once we selected the words from title and body we added them as features while adding '_title' or '_body' to differentiate words from the body and from the title.

Finally we run several classification models on the pre-processed data such as: a baseline model (predicting the most frequent outcome i.e. no fake news), a Logisitic Regression with a threshold at 0.5, a Classification Tree, a Random Forest and finally a Gradient Boosting model.

- **Why have you chosen this model?**

TF-IDF is widely use as a NLP pre-processing method. This model often shows good performance, is quite easy to understand and interpretable. We could also easily compare the performance of this model to the bag of words models, as they are both pre-processing methods.

We have chosen to combine our TF-IDF model with a Logisitic Regression with a threshold at 0.5, a Classification Tree, a Random Forest and finally a Gradient Boosting model, because these are common classification models. We first tried relatively simple models as a Logistic Regression and a Classification Tree because we knew they would be shorter to run in comparison to the Random Forest and Gradient Boosting (which are known for having a better performance in general).

- **How have you selected any tuning parameters for your models (a.k.a hyperparameters)?** For the Logistic Regression, we chose a threshold of 0.5 as we decided that the classes 'fake news' and 'reals news' have the same weight. We performed cross-validation on the validation set to tune the hyper-parameters of the Classification Tree, the Random Forest and the Gradient Boosting models. We performed it on the validation set which is smaller, then the cross-validation was less time and memory consuming. We chose the parameters such as their values maximize the f1 score.

- **How well does your model perform and how have you measured that?** We measured the performance by running our model on the testing set considering several metrics : f1-score, accuracy, TPR and FPR. However, we put our focus especially on the f1-score that we considered as the most important. The performance metrics are given by the table below. The model performed quite well: the f1-score is up to 0.651, the baseline for accuracy is 0.716 and our TF-IDF models have an accuracy between 0.716 and 0.834. The model performing the best is the Logistic Regression, with the highest f1 score and accuracy. This might be due to the patterns of the data, but we might have improved the performance of the other models with a more precise parameters tuning. Indeed, cross validation was very long to run, as we had 61351 columns (words), because it is hard to filter the words with the TF-IDF method. We did our best to have the best hyperparameters with the ressources that we have, in terms of memory and time.
  This model performed better than bag-of-words, but we tried to improve these results by using more complex models.

- **Are there any significant limitations to your modeling methods (i.e. Where, when, how, and why can your model fail?)** The main limitation of the tf-idf model is that it takes a lot of time to run. The final data set after preprocessing using tf-idf had 61351 columns (words), because unlike bag of words, it is difficult to filter the words used with tf-idf. For the same reason, it was also very long to perform cross-validation to fit the hyper-parameters, so the tuning could have been more precise if we had more time, and maybe the performance could have been better.

[4]:
```
#@title
import pandas as pd
comparison_data = {'Baseline model': ['{:.3f}'.format(0.000),
                                      '{:.4f}'.format(0.716),
                                      '{:.3f}'.format(0.000),
                                    '{:.4f}'.format(0.000)],
                   'Logistic Regression': ['{:.3f}'.format(0.651),
```

```
                                        '{:.4f}'.format(0.834),
                                        '{:.3f}'.format(0.544),
                                        '{:.4f}'.format(0.050)],
                    'Classification Tree': ['{:.3f}'.format(0.494),
                                            '{:.4f}'.format(0.771),
                                            '{:.3f}'.format(0.395),
                                             '{:.4f}'.format(0.080)],
                    'Random Forest': ['{:.3f}'.format(0.563),
                                      '{:.4f}'.format(0.817),
                                      '{:.3f}'.format(0.416),
                                      '{:.4f}'.format(0.024)],
                    'Gradient Boosting': ['{:.3f}'.format(0.337),
                                          '{:.4f}'.format(0.765),
                                          '{:.3f}'.format(0.210),
                                          '{:.4f}'.format(0.014)]}

comparison_table = pd.DataFrame(data=comparison_data,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[4]:

	Baseline model	Logistic Regression	Classification Tree	Random Forest	\
f1-score	0.000	0.651	0.494	0.563	
accuracy	0.7160	0.8340	0.7710	0.8170	
TPR	0.000	0.544	0.395	0.416	
FPR	0.0000	0.0500	0.0800	0.0240	

	Gradient Boosting
f1-score	0.337
accuracy	0.7650
TPR	0.210
FPR	0.0140

Performance after adding the non-text features - We tried several models by adding the non-text related features to the columns of our dataset and ran the different models. - We got the following performance as a result:

[5]:
```
#@title
import pandas as pd
comparison_data = {'Baseline model': ['{:.3f}'.format(0.000),
                                      '{:.4f}'.format(0.716),
                                      '{:.3f}'.format(0.000),
                                     '{:.4f}'.format(0.000)],
                   'Logistic Regression': ['{:.3f}'.format(0.665),
                                      '{:.4f}'.format(0.84),
                                      '{:.3f}'.format(0.559),
                                      '{:.4f}'.format(0.048)],
                   'Classification Tree': ['{:.3f}'.format(0.681),
```

```
                                            '{:.4f}'.format(0.823),
                                            '{:.3f}'.format(0.664),
                                             '{:.4f}'.format(0.114)],
                    'Random Forest': ['{:.3f}'.format(0),
                                      '{:.4f}'.format(0),
                                      '{:.3f}'.format(0),
                                      '{:.4f}'.format(0)],
                    'Gradient Boosting': ['{:.3f}'.format(0),
                                          '{:.4f}'.format(0.716),
                                          '{:.3f}'.format(0),
                                          '{:.4f}'.format(0)]}

comparison_table = pd.DataFrame(data=comparison_data,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[5]:

	Baseline model	Logistic Regression	Classification Tree	Random Forest	\
f1-score	0.000	0.665	0.681	0.000	
accuracy	0.7160	0.8400	0.8230	0.0000	
TPR	0.000	0.559	0.664	0.000	
FPR	0.0000	0.0480	0.1140	0.0000	

	Gradient Boosting
f1-score	0.000
accuracy	0.7160
TPR	0.000
FPR	0.0000

The TF-IDF model is overall performing better when we include the non-text features. For the logistic regression model, there is a slight improvement, but for the other models, classification tree, random forest and Gradient Boosting, the improvement is significant.

The hyperparameters for these models were selected by cross-validation on the f1-score with GridSearchCV, used on the validation set which is smaller and more manageable.

3.0.3 Word2vec

- **How do these models work?**

  In this project, we explored two approaches for Word2Vec:

  - Build our own Word2Vec model from scratch using FakeNewsNet (the dataset we use in this project)
  - Use pre-trained Word2Vec model which already trained on Google News

  Word2Vec is a NLP technique to "vectorize" words. In the vectorizing process every word in a corpus of text (the title or the body of our articles) will be translated into a vector also called embedding with values in the range [-1,1]. For example, the word "health" might be translated into [-0.91, 0.50, 0.31, -0.02, 0.61].

To decide the vectorization for every word the goal of this technique will be to minimize the distance between the vectors of words which usually appear together will be small. For instance, the distance between "health" and "insurance" will be smaller than the one between "health" and "truck".

To build this model a corpus of text is needed to determine which words appear in similar contexts and assign embeddings close to each other. In this we train one model with the titles in the FakeNewsNet dataset and another with the bodies. We will see later how pre-trained models can also be used.

Once we have our Word2Vec model trained we can use it to translate every document (a title or a body) into a series of embeddings, one for every word. A simple way to use these embeddings to perform a real/fake classification is to compute the mean of the embeddings of all words in a document and use the mean of those embeddings as features for classification models such as Logistic Regression, CART, Random Forest and Boosting.

The other option we will explore with word2vec is to use a pretrained model to generate the embeddings (instead of our own model) and use the embeddings as inputs of different Neural Networks.

- **Why have you chosen these models?**

  Build Word2Vec model from scratch:

  Word2Vec is a very popular and useful NLP technique. The first model we built from scratch is simple but requires longer training time.

  Additionally, this model is a good prologue for the Doc2vec technique we will discuss in the next section.

  Logistic Regression, CART, Random Forest and Boosting have been chosen because they are the most common alternatives for classification with this kind of inputs.

  Word2Vec Google News pre-trained model:

  Aside from traning our own Word2Vec model, due to our news dataset, we also tried using a popular pre-trained Word2Vec model which was trained on Google News dataset, covering approximately 100 billions words. Such a model can take hours to train, but since it's already available, downloading and loading it with Gensim only takes minutes. After feeding our news data into the model, it will produce 300-dimensional vector embedding for each word.

  We then used its output in the embedding layer of a Long Short Term Memory (LSTM) and it is a common Neural Network model used in NLP. The LSTM differs additionally by the so-called cell state. This represents a long-term memory, which first forgets a part by multiplication with the previous cell state and then, if necessary, multiplies another, newer part to the memory. This means that there is not only a short-term memory through the loop, but also a long-term memory. Since language is often very complex and not every sentence unfolds its meaning from left to right, we also tried a bidirectional LSTM.

- **How have you selected any tuning parameters for your models (a.k.a hyperparameters)?**

  Once again cross-validation was used to select the parameters for CART, Random Forest and Boosting maximizing f1, while 0.5 was the selected threshold for Logistic Regression.

For Google News pre-trained Word2Vec LSTM and bidirectional LSTM models, bacth size, number of epochs and number of Neural Network layers are decided through cross-validation. Because this is a binary classification problem, we are using sigmoid as activation function, binary cross entropy as loss function, adam as optimizaation algorithm.

LSTM:

```
model_word2vec = Sequential()
model_word2vec.add(embedding_layer)
model_word2vec.add(LSTM(32,  dropout=0.2, recurrent_dropout=0.25))
model_word2vec.add(Dense(1, activation='sigmoid'))
model_word2vec.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model_word2vec.summary())
```

Bidirectional LSTM:

```
model_word2vec3 = Sequential()
model_word2vec3.add(embedding_layer)
model_word2vec3.add(Bidirectional(LSTM(100)))
model_word2vec3.add(Dropout(0.3))
model_word2vec3.add(Dense(1, activation='sigmoid'))
model_word2vec3.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model_word2vec3.summary())
```

- **How well does your model perform and how have you measured that?**

  To evaluate the model performance, we also look at F1 score, TPR (Recall), FPR and Accuracy for each model.

  For the Logistic Regression, CART, Random Forest and Boosting models, we have obtained slightly higher scores than the ones obtained with bag of words and similar ones while working with TF_IDF when we apply wod2vec to the bodies of the news, while word2vec applied on the titles performs similarly to bag of words.

  Google News pre-trained Word2Vec model is performing slightly better than FakeNewsNet trained Word2Vec model.

FakeNewsNet trained Word2Vec model performance:

For news titles

[6]:
```
#@title
comparison_data_w2v_t = {'Baseline model': ['{:.3f}'.format(0.000),
                                            '{:.4f}'.format(0.716),
                                            '{:.3f}'.format(0.000),
                                          '{:.4f}'.format(0.000)],
                         'Logistic Regression': ['{:.3f}'.format(0.513),
                                            '{:.4f}'.format(0.783),
                                            '{:.3f}'.format(0.401),
                                            '{:.4f}'.format(0.065)],
                         'Classification Tree': ['{:.3f}'.format(0.479),
                                                 '{:.4f}'.format(0.783),
                                                 '{:.3f}'.format(0.452),
```

```
                                                '{:.4f}'.format(0.173)],
                    'Random Forest': ['{:.3f}'.format(0.477),
                                      '{:.4f}'.format(0.77),
                                      '{:.3f}'.format(0.353),
                                      '{:.4f}'.format(0.051)],
                    'Gradient Boosting': ['{:.3f}'.format(0.466),
                                          '{:.4f}'.format(0.78),
                                          '{:.3f}'.format(0.338),
                                          '{:.4f}'.format(0.06)]}

comparison_table = pd.DataFrame(data=comparison_data_w2v_t,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[6]:

	Baseline model	Logistic Regression	Classification Tree	Random Forest	\
f1-score	0.000	0.513	0.479	0.477	
accuracy	0.7160	0.7830	0.7830	0.7700	
TPR	0.000	0.401	0.452	0.353	
FPR	0.0000	0.0650	0.1730	0.0510	

	Gradient Boosting
f1-score	0.466
accuracy	0.7800
TPR	0.338
FPR	0.0600

For news bodies

[7]:
```
#@title
comparison_data_w2v_b = {'Baseline model': ['{:.3f}'.format(0.000),
                                            '{:.4f}'.format(0.716),
                                            '{:.3f}'.format(0.000),
                                          '{:.4f}'.format(0.000)],
                    'Logistic Regression': ['{:.3f}'.format(0.627),
                                            '{:.4f}'.format(0.825),
                                            '{:.3f}'.format(0.519),
                                            '{:.4f}'.format(0.054)],
                    'Classification Tree': ['{:.3f}'.format(0.519),
                                            '{:.4f}'.format(0.75),
                                            '{:.3f}'.format(0.474),
                                             '{:.4f}'.format(0.14)],
                    'Random Forest': ['{:.3f}'.format(0.541),
                                      '{:.4f}'.format(0.807),
                                      '{:.3f}'.format(0.4),
                                      '{:.4f}'.format(0.032)],
                    'Gradient Boosting': ['{:.3f}'.format(0.538),
                                          '{:.4f}'.format(0.812),
```

```
                                                '{:.3f}'.format(0.387),
                                                '{:.4f}'.format(0.026)]}

comparison_table = pd.DataFrame(data=comparison_data_w2v_b,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[7]:

	Baseline model	Logistic Regression	Classification Tree	Random Forest	Gradient Boosting
f1-score	0.000	0.627	0.519	0.541	0.538
accuracy	0.7160	0.8250	0.7500	0.8070	0.8120
TPR	0.000	0.519	0.474	0.400	0.387
FPR	0.0000	0.0540	0.1400	0.0320	0.0260

Google News pre-trained Word2Vec model performance:

For news titles

[8]:
```
#@title

comparison_data_w2v_b = {'Baseline model': ['{:.2f}'.format(0.000),
                                            '{:.2f}'.format(0.716),
                                            '{:.2f}'.format(0.000),
                                          '{:.2f}'.format(0.000)],
                         'LSTM': ['{:.2f}'.format(0.54),
                                            '{:.2f}'.format(0.78),
                                            '{:.2f}'.format(0.47),
                                            '{:.2f}'.format(0.10)],
                         'Bidirectional LSTM': ['{:.2f}'.format(0.57),
                                                '{:.2f}'.format(0.74),
                                                '{:.2f}'.format(0.61),
                                                 '{:.2f}'.format(0.21)]}

comparison_table = pd.DataFrame(data=comparison_data_w2v_b,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[8]:

	Baseline model	LSTM	Bidirectional LSTM
f1-score	0.00	0.54	0.57
accuracy	0.72	0.78	0.74
TPR	0.00	0.47	0.61
FPR	0.00	0.10	0.21

News bodies

[9]:
```
#@title

comparison_data_w2v_b = {'Baseline model': ['{:.2f}'.format(0.000),
                                            '{:.2f}'.format(0.716),
                                            '{:.2f}'.format(0.000),
                                          '{:.2f}'.format(0.000)],
                         'LSTM': ['{:.2f}'.format(0.59),
                                            '{:.2f}'.format(0.80),
                                            '{:.2f}'.format(0.48),
                                            '{:.2f}'.format(0.06)],
                         'Bidirectional LSTM': ['{:.2f}'.format(0.53),
                                                '{:.2f}'.format(0.73),
                                                '{:.2f}'.format(0.54),
                                                 '{:.2f}'.format(0.19)]}

comparison_table = pd.DataFrame(data=comparison_data_w2v_b,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[9]:

	Baseline model	LSTM	Bidirectional LSTM
f1-score	0.00	0.59	0.53
accuracy	0.72	0.80	0.73
TPR	0.00	0.48	0.54
FPR	0.00	0.06	0.19

- **Are there any significant limitations to your modeling methods (i.e. Where, when, how, and why can your model fail?)**

  In the case of the word2vec model trained with the FakeNewsNet data, the training corpus is small to train an accurate model. That is the reason why we later opted for a pre-trained word2vec. At the same time, one common limitation for all models where we use Logistic Regression, CART, Random Forest and Boosting, is that during the process we loose the information provided by word ordering. This problem is solved when we apply models such as LSTM.

  However, Neural Network is prone to overfitting, so we plot graphs of accuracy and loss vs number of epochs to analyze if the model is overfitting to training set.

  LSTM model (left) and Bidirectional LSTM model (right):

As shown above, difference in accuracy between training and testing sets is larger in Bidirectional LSTM model, same happened in the loss between training and testing sets. Therefore, LSTM model is less overfitting than Bidirectional LSTM model. Because of this, we used output from LSTM model to concatenate with other non-text features.

- **Performance after adding the non-text features**

We tried several models by concatenating the non-text related features and the outcomes of the Word2Vec models based on news body and on news title.We got the following performance as a result:

FakeNewsNet trained Word2Vec model performance

[10]:
```
#@title
import pandas as pd
comparison_data = {'Baseline model': ['{:.3f}'.format(0.000),
                                      '{:.3f}'.format(0.716),
                                      '{:.3f}'.format(0.000),
                                     '{:.3f}'.format(0.000)],
                   'Logistic Regression': ['{:.3f}'.format(0.624),
                                      '{:.3f}'.format(0.818),
                                      '{:.3f}'.format(0.53),
                                      '{:.3f}'.format(0.068)],
                   'Classification Tree': ['{:.3f}'.format(0.755),
                                           '{:.3f}'.format(0.871),
                                           '{:.3f}'.format(0.7),
                                            '{:.3f}'.format(0.062)],
                   'Random Forest': ['{:.3f}'.format(0.773),
                                     '{:.3f}'.format(0.88),
                                     '{:.3f}'.format(0.721),
                                     '{:.3f}'.format(0.057)],
                   'Gradient Boosting': ['{:.3f}'.format(0.788),
                               '{:.3f}'.format(0.89),
                               '{:.3f}'.format(0.721),
                               '{:.3f}'.format(0.043)]}

comparison_table = pd.DataFrame(data=comparison_data,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[10]:

	Baseline model	Logistic Regression	Classification Tree	Random Forest	\
f1-score	0.000	0.624	0.755	0.773	
accuracy	0.716	0.818	0.871	0.880	
TPR	0.000	0.530	0.700	0.721	
FPR	0.000	0.068	0.062	0.057	

	Gradient Boosting
f1-score	0.788
accuracy	0.890
TPR	0.721
FPR	0.043

Google News pre-trained Word2Vec model performance

[11]:
```
#@title
import pandas as pd
comparison_data = {'Baseline model': ['{:.3f}'.format(0.000),
                                      '{:.3f}'.format(0.716),
                                      '{:.3f}'.format(0.000),
                                     '{:.3f}'.format(0.000)],
                   'Logistic Regression': ['{:.3f}'.format(0.356),
                                      '{:.3f}'.format(0.698),
                                      '{:.3f}'.format(0.293),
                                      '{:.3f}'.format(0.141)],
                   'Classification Tree': ['{:.3f}'.format(0.651),
                                           '{:.3f}'.format(0.812),
                                           '{:.3f}'.format(0.616),
                                            '{:.3f}'.format(0.11)],
                   'Random Forest': ['{:.3f}'.format(0.633),
                                     '{:.3f}'.format(0.809),
                                     '{:.3f}'.format(0.579),
                                     '{:.3f}'.format(0.1)],
                   'Gradient Boosting': ['{:.3f}'.format(0.699),
                              '{:.3f}'.format(0.846),
                              '{:.3f}'.format(0.63),
                              '{:.3f}'.format(0.0169)]}

comparison_table = pd.DataFrame(data=comparison_data,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[11]:

	Baseline model	Logistic Regression	Classification Tree	Random Forest	\
f1-score	0.000	0.356	0.651	0.633	
accuracy	0.716	0.698	0.812	0.809	
TPR	0.000	0.293	0.616	0.579	
FPR	0.000	0.141	0.110	0.100	

	Gradient Boosting
f1-score	0.699
accuracy	0.846
TPR	0.630
FPR	0.017

3.0.4 GloVe

- **How do these models work?**

  A well-known model that learns vectors or words from their co-occurrence information, i.e. how frequently they appear together in large text corpora, is GlobalVectors (GloVe).

- **Why have you chosen these models?**

  Similar to Word2Vec, GloVe model also assigns vector embedding for each word. The two

models differ in the way they are trained, and hence lead to word vectors with subtly different properties. GloVe model is based on leveraging global word to word co-occurance counts leveraging the entire corpus. Word2vec on the other hand leverages co-occurance within local context (neighbouring words).

After getting the word embeddings, we feed the vector into Neural Network - LSTM and bidirectional LSTM.

- **How have you selected any tuning parameters for your models (a.k.a hyperparameters)?**

  The hyperparameter tuning for Neural Network is identical as mentioned under Word2Vec section.

- **How well does your model perform and how have you measured that?**

  To evaluate the model performance, we also look at F1 score, TPR (Recall), FPR and Accuracy for each model.

For news titles

[12]:
```
#@title

comparison_data_w2v_b = {'Baseline model': ['{:.2f}'.format(0.000),
                                            '{:.2f}'.format(0.716),
                                            '{:.2f}'.format(0.000),
                                          '{:.2f}'.format(0.000)],
                         'LSTM': ['{:.2f}'.format(0.54),
                                            '{:.2f}'.format(0.79),
                                            '{:.2f}'.format(0.46),
                                            '{:.2f}'.format(0.08)],
                         'Bidirectional LSTM': ['{:.2f}'.format(0.56),
                                                '{:.2f}'.format(0.74),
                                                '{:.2f}'.format(0.49),
                                                 '{:.2f}'.format(0.14)]}

comparison_table = pd.DataFrame(data=comparison_data_w2v_b,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[12]:

	Baseline model	LSTM	Bidirectional LSTM
f1-score	0.00	0.54	0.56
accuracy	0.72	0.79	0.74
TPR	0.00	0.46	0.49
FPR	0.00	0.08	0.14

For news bodies

[13]:
```
#@title

comparison_data_w2v_b = {'Baseline model': ['{:.2f}'.format(0.000),
```

```
                                        '{:.2f}'.format(0.716),
                                        '{:.2f}'.format(0.000),
                                      '{:.2f}'.format(0.000)],
                    'LSTM': ['{:.2f}'.format(0.59),
                                        '{:.2f}'.format(0.79),
                                        '{:.2f}'.format(0.52),
                                         '{:.2f}'.format(0.09)],
                    'Bidirectional LSTM': ['{:.2f}'.format(0.53),
                                        '{:.2f}'.format(0.76),
                                        '{:.2f}'.format(0.51),
                                         '{:.2f}'.format(0.15)]}

comparison_table = pd.DataFrame(data=comparison_data_w2v_b,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[13]:

	Baseline model	LSTM	Bidirectional LSTM
f1-score	0.00	0.59	0.53
accuracy	0.72	0.79	0.76
TPR	0.00	0.52	0.51
FPR	0.00	0.09	0.15

- **Are there any significant limitations to your modeling methods (i.e. Where, when, how, and why can your model fail?)**

  Similar to Word2Vec, we used Neural Network for GloVe model and it is prone to overfitting.

  LSTM model (left) and Bidirectional LSTM model (right):

As shown above, difference in accuracy between training and testing sets is larger in Bidirectional LSTM model, same happened in the loss between training and testing sets. Therefore, LSTM model is less overfitting than Bidirectional LSTM model. Because of this, we used output from LSTM model to concatenate with other non-text features.

- **Performance after adding the non-text features**

  We tried several models by concatenating the non-text related features and the outcomes of the GloVe models based on news body and on news title. We got the following performance as a result:

[14]:
```
#@title
import pandas as pd
comparison_data = {'Baseline model': ['{:.3f}'.format(0.000),
                                        '{:.3f}'.format(0.716),
                                        '{:.3f}'.format(0.000),
                                      '{:.3f}'.format(0.000)],
                    'Logistic Regression': ['{:.3f}'.format(0.298),
```

```
                                        '{:.3f}'.format(0.691),
                                        '{:.3f}'.format(0.23),
                                        '{:.3f}'.format(0.126)],
                    'Classification Tree': ['{:.3f}'.format(0.518),
                                            '{:.3f}'.format(0.774),
                                            '{:.3f}'.format(0.427),
                                             '{:.3f}'.format(0.089)],
                    'Random Forest': ['{:.3f}'.format(0.541),
                                      '{:.3f}'.format(0.783),
                                      '{:.3f}'.format(0.449),
                                      '{:.3f}'.format(0.084)],
                    'Gradient Boosting': ['{:.3f}'.format(0.686),
                                '{:.3f}'.format(0.834),
                                '{:.3f}'.format(0.638),
                                '{:.3f}'.format(0.088)]}

comparison_table = pd.DataFrame(data=comparison_data,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[14]:

	Baseline model	Logistic Regression	Classification Tree	Random Forest	\
f1-score	0.000	0.298	0.518	0.541	
accuracy	0.716	0.691	0.774	0.783	
TPR	0.000	0.230	0.427	0.449	
FPR	0.000	0.126	0.089	0.084	

	Gradient Boosting
f1-score	0.686
accuracy	0.834
TPR	0.638
FPR	0.088

3.0.5 Doc2Vec

- **How do these models work?**

  Doc2Vec is another NLP technique for representing documents of text as vectors. In contrast with methods such as word2vec, doc2vec translate directly a text into an embedding.

  To obtain this doc embedding, the doc2vec model is trained using 2 inputs:

  – **Word vectors**: this is similar to the output of word2vec, obtaining one vector for every word in the document.

  – **Paragraph ID**: the paragraph ID is obtained following a similar process to the one applied for every word to get a word embedding. This embedding is produced taking random groups of words in the paragraph. This way, the model includes more contextual information about the whole text than in the case of applying just word2vec.

Once computed the word vectors and the paragraph id for every document, both are the input of a Neural Network which summarizes all the document into one embedding.

Once we have obtained the embedding for a document, which in our case could be the title or the body of the article, we can apply models such as Logistic Regression, CART, Random Forest or Boosting.

Despite there are also pretrained Doc2Vec models, in contrast with word2vec, the performance of these models is usually poor. Therefore we have trained two different Doc2Vec models: one to work with titles and another for bodies.

- **Why have you chosen these models?**

  Doc2Vec is a great model to work with pieces of text, since it combines the advantages of word2vec with a higher weight for contextual information. Additionally, its output is suitable for use with traditional classification models such as logistic regression or decision trees.

- **How have you selected any tuning parameters for your models (a.k.a hyperparameters)?**

  Once again cross-validation was used to select the parameters for CART, Random Forest and Boosting maximizing f1, while 0.5 was the selected threshold for Logistic Regression.

- **How well does your model perform and how have you measured that?**

  To evaluate the model performance, we also look at F1 score, TPR (Recall), FPR and Accuracy for each model.

  For the Logistic Regression, CART, Random Forest and Boosting models, we have obtained worse results than in the most similar model: mean word2vec, for most of the alternatives. The main reason is that Doc2Vec performs better than mean word2vec especially when it is trained using a lot of data. Since the data availible in FakeNewsNet is not big enough the performance of mean word2vec was superior.

  Performance in titles:

[15]:
```
#@title
comparison_data_d2v_t = {'Baseline model': ['{:.3f}'.format(0.000),
                                            '{:.4f}'.format(0.716),
                                            '{:.3f}'.format(0.000),
                                          '{:.4f}'.format(0.000)],
                         'Logistic Regression': ['{:.3f}'.format(0.392),
                                                 '{:.4f}'.format(0.749),
                                                 '{:.3f}'.format(0.285),
                                                 '{:.4f}'.format(0.066)],
                         'Classification Tree': ['{:.3f}'.format(0.396),
                                                      '{:.4f}'.format(0.647),
                                                      '{:.3f}'.format(0.407),
                                                       '{:.4f}'.format(0.257)],
                         'Random Forest': ['{:.3f}'.format(0.433),
                                           '{:.4f}'.format(0.755),
                                           '{:.3f}'.format(0.329),
                                           '{:.4f}'.format(0.076)],
```

```
                  'Gradient Boosting': ['{:.3f}'.format(0.435),
                                        '{:.4f}'.format(0.765),
                                        '{:.3f}'.format(0.342),
                                        '{:.4f}'.format(0.152)]}

comparison_table = pd.DataFrame(data=comparison_data_d2v_t,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[15]:

	Baseline model	Logistic Regression	Classification Tree	Random Forest	\
f1-score	0.000	0.392	0.396	0.433	
accuracy	0.7160	0.7490	0.6470	0.7550	
TPR	0.000	0.285	0.407	0.329	
FPR	0.0000	0.0660	0.2570	0.0760	

	Gradient Boosting
f1-score	0.435
accuracy	0.7650
TPR	0.342
FPR	0.1520

Performance in news bodies:

[16]:
```
#@title
comparison_data_d2v_b = {'Baseline model': ['{:.3f}'.format(0.000),
                                            '{:.4f}'.format(0.716),
                                            '{:.3f}'.format(0.000),
                                          '{:.4f}'.format(0.000)],
                         'Logistic Regression': ['{:.3f}'.format(0.534),
                                            '{:.4f}'.format(0.789),
                                            '{:.3f}'.format(0.425),
                                            '{:.4f}'.format(0.066)],
                         'Classification Tree': ['{:.3f}'.format(0.509),
                                              '{:.4f}'.format(0.735),
                                              '{:.3f}'.format(0.482),
                                               '{:.4f}'.format(0.164)],
                         'Random Forest': ['{:.3f}'.format(0.545),
                                           '{:.4f}'.format(0.795),
                                           '{:.3f}'.format(0.432),
                                           '{:.4f}'.format(0.06)],
                         'Gradient Boosting': ['{:.3f}'.format(0.515),
                                              '{:.4f}'.format(0.785),
                                              '{:.3f}'.format(0.435),
                                              '{:.4f}'.format(0.09)]}

comparison_table = pd.DataFrame(data=comparison_data_d2v_b,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
```

```
comparison_table
```

[16]:

	Baseline model	Logistic Regression	Classification Tree	Random Forest	Gradient Boosting
f1-score	0.000	0.534	0.509	0.545	0.515
accuracy	0.7160	0.7890	0.7350	0.7950	0.7850
TPR	0.000	0.425	0.482	0.432	0.435
FPR	0.0000	0.0660	0.1640	0.0600	0.0900

- **Are there any significant limitations to your modeling methods (i.e. Where, when, how, and why can your model fail?)**

  As mentioned in the previous performance of doc2vec trained in a small text corpus is usually low, what makes it especially limited when we use it on the titles.

- **Performance after adding the non-text features**

  We tried several models by concatenating the non-text related features and the outcomes of the Word2Vec models based on news body and on news title.We got the following performance as a result:

[17]:
```
#@title
import pandas as pd
comparison_data = {'Baseline model': ['{:.3f}'.format(0.000),
                                      '{:.3f}'.format(0.716),
                                      '{:.3f}'.format(0.000),
                                    '{:.3f}'.format(0.000)],
                   'Logistic Regression': ['{:.3f}'.format(0.624),
                                      '{:.3f}'.format(0.785),
                                      '{:.3f}'.format(0.382),
                                      '{:.3f}'.format(0.056)],
                   'Classification Tree': ['{:.3f}'.format(0.554),
                                         '{:.3f}'.format(0.797),
                                         '{:.3f}'.format(0.442),
                                          '{:.3f}'.format(0.062)],
                   'Random Forest': ['{:.3f}'.format(0.542),
                                     '{:.3f}'.format(0.811),
                                     '{:.3f}'.format(0.393),
                                     '{:.3f}'.format(0.023)],
                   'Gradient Boosting': ['{:.3f}'.format(0.436),
                               '{:.3f}'.format(0.755),
                               '{:.3f}'.format(0.333),
                               '{:.3f}'.format(0.077)]}
```

```
comparison_table = pd.DataFrame(data=comparison_data,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[17]:

	Baseline model	Logistic Regression	Classification Tree	Random Forest	Gradient Boosting
f1-score	0.000	0.624	0.554	0.542	0.436
accuracy	0.716	0.785	0.797	0.811	0.755
TPR	0.000	0.382	0.442	0.393	0.333
FPR	0.000	0.056	0.062	0.023	0.077

3.0.6 Fasttext

- **How do these models work and why have you chosen these models?**

  Fasttext is a library for learning of word embeddings and text classification created by Facebook's AI Research lab.

  Vectors are given to sentences by averaging the embeddings of word-N-gram (number of characters in a row taken into account). Then , for the supervised classification, a multinomial logistic regression is used (here a simple logistic regression since there are only 2 classes) with the sentence vectors as the features.

  Vectors are given by using a model derived from the skipgram model (taking into account subword information to create the vector). A skipgram model is a neural network model using 2 layers: one of 300 neurons with linear activation function, and one of 10,000 neurons with softmax activation function. This model is using a word to predict the probabilities for each given word of the vocabulary (10,000 words here) to be nearby this word.

  The model is trained on Wikipedia data. Its specificity is that the fastText classification model is fast because of its relative simplicity (no deep learning but a linear classifier).

  We chose this model because it is well-know for its speed (while having a good performance).

- **How have you selected any tuning parameters for your models (a.k.a hyperparameters)?** There was no hyperparameter to tune, but we could select the loss function and the number of words in a row to take into account. For the loss, the model allows to choose between 'Negative sampling', 'Hierarchical Softmax' and 'Softmax'. We chose negative sampling as the softmax and sigmoid are equivalent in case of binary classification. For the number of words, we tried with different parameters on the testing set and *wordNgrams=1* was giving the best performance metric.

- **How well does your model perform and how have you measured that?** As for the other models, we focused on the f1-score but also took into acount some other performance metrics (accuracy, TPR and FPR). This mdoel performs well in comparison to the other models, especially on body data (on larger datasets). In addition to that, the mdoel is

extremely fast to run, which is a parameter we take into account, especially if we want to scale the model.

- **Are there any significant limitations to your modeling methods (i.e. Where, when, how, and why can your model fail?)** The main limitation of the model is its difficult interpretability, as it is a pretrained model.

[18]:
```
#@title
import pandas as pd
comparison_data = {'Title': ['{:.3f}'.format(0.628),
                             '{:.3f}'.format(0.802),
                             '{:.3f}'.format(0.586),
                            '{:.3f}'.format(0.112)],
                   'Body': ['{:.3f}'.format(0.646),
                             '{:.3f}'.format(0.824),
                             '{:.3f}'.format(0.564),
                             '{:.3f}'.format(0.073)]}

comparison_table = pd.DataFrame(data=comparison_data,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[18]:

	Title	Body
f1-score	0.628	0.646
accuracy	0.802	0.824
TPR	0.586	0.564
FPR	0.112	0.073

Performance after adding the non-text features - We tried several models by concatenating the non-text related features to the columns of our dataset and the outcomes of the fastText model on the body and on the title. - We got the following performance as a result:

[19]:
```
#@title
import pandas as pd
comparison_data = {'Baseline model': ['{:.3f}'.format(0.000),
                             '{:.3f}'.format(0.716),
                             '{:.3f}'.format(0.000),
                            '{:.3f}'.format(0.000)],
                   'Logistic Regression': ['{:.3f}'.format(0.633),
                             '{:.3f}'.format(0.804),
                             '{:.3f}'.format(0.595),
                             '{:.3f}'.format(0.113)],
                   'Classification Tree': ['{:.3f}'.format(0.714),
                                '{:.3f}'.format(0.841),
                                '{:.3f}'.format(0.695),
                                 '{:.3f}'.format(0.1)],
                   'Random Forest': ['{:.3f}'.format(0.746),
                            '{:.3f}'.format(0.864),
```

```
                                    '{:.3f}'.format(0.704),
                                    '{:.3f}'.format(0.073)],
                    'Gradient Boosting': ['{:.3f}'.format(0.771),
                                '{:.3f}'.format(0.878),
                                '{:.3f}'.format(0.723),
                                '{:.3f}'.format(0.061)]}

comparison_table = pd.DataFrame(data=comparison_data,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[19]:

	Baseline model	Logistic Regression	Classification Tree	Random Forest	Gradient Boosting
f1-score	0.000	0.633	0.714	0.746	0.771
accuracy	0.716	0.804	0.841	0.864	0.878
TPR	0.000	0.595	0.695	0.704	0.723
FPR	0.000	0.113	0.100	0.073	0.061

For each model we did some cross-validation with GridSearchCV on the f1-score to select the hyperparameters, except for the XGBoost model for which we used the validation set.

We notice that the model performing best is the XGBoost model. However, even if the fastText model performs better than the bag of words model for the text only, it does not perform as well when non-text features are included.

3.0.7 Assembling of the TF-IDF and Fast-text models

- **How does this model work?**

  We wanted to improve the performance of our models by combining the results of both of them. We ran both models on the text separetely, and we considered the output vectors to make new predictions. For the TF-IDF model, after text preprocessing, we took the output from the Logistic regression model, as it was the most performing one. We had three output vectors: 1 from the TF-IDF + Logistic regression model, and 2 from the Fasttext model (one after running a model on the titles of the article, and the other on the content). We used these output vectors as new features for supervised learning models. The process and the description of the TF-IDF and Fasttext models are described above.
  We applied 4 classification models: Logistic Regression, Classification Tree, Random Forest, Gradient Boosting.

- **Why have you chosen these models?**

  We wanted to combine the performance of our models to improve it. We chose the most performant models that we had, based mainly on the f1 score, and also both models had a very different way of working. The TF-IDF model attributes a vector to each vector, based on

two metrics: the term frequency in the article and the inverse document frequency, taking into account the frequency of the word in multiple documents. The fast text model is pre-trained on Wikipedia data, attribute a vector to each sentence and then use a logistic regression and a neural network to make predictions. We thought it would be interesting to combine two different models to take advantage of the strength of both models.

- **How have you selected any tuning parameters for your models (a.k.a hyperparameters)?**
  To get the output of each NLP model (TF-IDF and Fasttext), we used the same method and outputs as before. The parameters tuning is described above for each model. Then, we performed hyperparameters tuning for the supervised learning models we applied. For the Logistic Regression, we chose a threshold of 0.5 as we decided that the classes 'fake news' and 'reals news' have the same weight. We performed cross-validation on the validation set to tune the hyper-parameters of the Classification Tree, the Random Forest and the Gradient Boosting models. We performed it on the validation set which is smaller, then the cross-validation was less time and memory consuming. We chose the parameters such as their values maximize the f1 score.

- **How well does your model perform and how have you measured that?**
  We measured the performance by running our model on the testing set considering several metrics : f1-score, accuracy, TPR and FPR. However, we put our focus especially on the f1-score that we considered as the most important. The performance metrics are given by the table below. The model performed well: the f1-score is up to 0.667 for the random forest model, the baseline for accuracy is 0.716 and our model have an accuracy between 0.802 and 0.815. The models performing the best is the Random Forest and the Logistic Ression, which have a f1 score and an accuracy very close. This model performed better than the TF-IDF alone, but it only improved slightly the performance of Fasttext.

- **Are there any significant limitations to your modeling methods (i.e. Where, when, how, and why can your model fail?)** The main limitations are the ones of each model respectively: running time for TF-IDF and interpretability for Fasttext. Because we combined two models, the resulting model is also less interpretable.

[20]:
```
import pandas as pd
comparison_data = {'Baseline model': ['{:.3f}'.format(0.000),
                                      '{:.4f}'.format(0.716),
                                      '{:.3f}'.format(0.000),
                                     '{:.4f}'.format(0.000)],
                   'Logistic Regression': ['{:.3f}'.format(0.666),
                                      '{:.4f}'.format(0.815),
                                      '{:.3f}'.format(0.649),
                                      '{:.4f}'.format(0.12)],
                   'Classification Tree': ['{:.3f}'.format(0.628),
                                           '{:.4f}'.format(0.802),
                                           '{:.3f}'.format(0.586),
                                            '{:.4f}'.format(0.112)],
                   'Random Forest': ['{:.3f}'.format(0.667),
                                     '{:.4f}'.format(0.814),
                                     '{:.3f}'.format(0.653),
```

```
                                    '{:.4f}'.format(0.122)],
                  'Gradient Boosting': ['{:.3f}'.format(0.628),
                                        '{:.4f}'.format(0.802),
                                        '{:.3f}'.format(0.586),
                                        '{:.4f}'.format(0.112)]}

comparison_table = pd.DataFrame(data=comparison_data,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[20]:

	Baseline model	Logistic Regression	Classification Tree	Random Forest	\
f1-score	0.000	0.666	0.628	0.667	
accuracy	0.7160	0.8150	0.8020	0.8140	
TPR	0.000	0.649	0.586	0.653	
FPR	0.0000	0.1200	0.1120	0.1220	

	Gradient Boosting
f1-score	0.628
accuracy	0.8020
TPR	0.586
FPR	0.1120

Performance after adding the non-text features - We tried several models by adding the non-text related features to the columns of our dataset and ran the different models. - We got the following performance as a result:

[21]:
```
import pandas as pd
comparison_data = {'Baseline model': ['{:.3f}'.format(0.000),
                                      '{:.4f}'.format(0.716),
                                      '{:.3f}'.format(0.000),
                                     '{:.4f}'.format(0.000)],
                   'Logistic Regression': ['{:.3f}'.format(0.668),
                                      '{:.4f}'.format(0.82),
                                      '{:.3f}'.format(0.636),
                                      '{:.4f}'.format(0.107)],
                   'Classification Tree': ['{:.3f}'.format(0.628),
                                        '{:.4f}'.format(0.802),
                                        '{:.3f}'.format(0.586),
                                         '{:.4f}'.format(0.112)],
                   'Random Forest': ['{:.3f}'.format(0.747),
                                     '{:.4f}'.format(0.865),
                                     '{:.3f}'.format(0.7),
                                     '{:.4f}'.format(0.069)],
                   'Gradient Boosting': ['{:.3f}'.format(0.68),
                                        '{:.4f}'.format(0.846),
                                        '{:.3f}'.format(0.575),
                                        '{:.4f}'.format(0.046)]}
```

```
comparison_table = pd.DataFrame(data=comparison_data,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[21]:

	Baseline model	Logistic Regression	Classification Tree	Random Forest	\
f1-score	0.000	0.668	0.628	0.747	
accuracy	0.7160	0.8200	0.8020	0.8650	
TPR	0.000	0.636	0.586	0.700	
FPR	0.0000	0.1070	0.1120	0.0690	

	Gradient Boosting
f1-score	0.680
accuracy	0.8460
TPR	0.575
FPR	0.0460

This model is overall performing better when we include the non-text features. There is a slight improvement for every models, except for Random Forest for which the improvement is significant. This is the most performant model for now.

The hyperparameters for these models were selected by cross-validation on the f1-score with GridSearchCV, used on the validation set which is smaller and more manageable.

4 Conclusion

4.1 Performance Table

As final recommendation, we compared performance of models which included both textual and non-textual features.

In all cases, Gradient Boosting has the best performance, so in the below table, we are showing the performance metrics of Gradient Boosting when both textual and non-textual features fed into the model with the textual features had been processed by different NLP techniques.

[22]:
```
#@title
import pandas as pd
comparison_data = {'Baseline model': ['{:.3f}'.format(0.000),
                                      '{:.4f}'.format(0.716),
                                      '{:.3f}'.format(0.000),
                                     '{:.4f}'.format(0.000)],
                   'Bag of words': ['{:.3f}'.format(0.779),
                                      '{:.4f}'.format(0.884),
                                      '{:.3f}'.format(0.715),
                                      '{:.4f}'.format(0.048)],
                   'TF-IDF': ['{:.3f}'.format(0.665),
                                          '{:.4f}'.format(0.84),
                                          '{:.3f}'.format(0.559),
```

```
                                          '{:.4f}'.format(0.048)],
                  'Word2Vec': ['{:.3f}'.format(0.788),
                               '{:.4f}'.format(0.890),
                               '{:.3f}'.format(0.721),
                               '{:.4f}'.format(0.043)],
                  'GloVe': ['{:.3f}'.format(0.686),
                                        '{:.4f}'.format(0.834),
                                        '{:.3f}'.format(0.638),
                                        '{:.4f}'.format(0.088)],
                  'Doc2Vec': ['{:.3f}'.format(0.624),
                                        '{:.4f}'.format(0.785),
                                        '{:.3f}'.format(0.382),
                                        '{:.4f}'.format(0.056)],
                  'Fasttext': ['{:.3f}'.format(0.771),
                                        '{:.4f}'.format(0.878),
                                        '{:.3f}'.format(0.723),
                                        '{:.4f}'.format(0.061)],
                  'TF-IDF + Fasttext': ['{:.3f}'.format(0.747),
                                        '{:.4f}'.format(0.865),
                                        '{:.3f}'.format(0.7),
                                        '{:.4f}'.format(0.069)]}

comparison_table = pd.DataFrame(data=comparison_data,␣
 ↪index=['f1-score','accuracy', 'TPR', 'FPR'])
comparison_table
```

[22]:

	Baseline model	Bag of words	TF-IDF	Word2Vec	GloVe	Doc2Vec	Fasttext	TF-IDF + Fasttext
f1-score	0.000	0.779	0.665	0.788	0.686	0.624	0.771	0.747
accuracy	0.7160	0.8840	0.8400	0.8900	0.8340	0.7850	0.8780	0.8650
TPR	0.000	0.715	0.559	0.721	0.638	0.382	0.723	0.700
FPR	0.0000	0.0480	0.0480	0.0430	0.0880	0.0560	0.0610	0.0690

4.2 Time Estimation for each NLP models

- **Bag of words:** ~40min each supervised learning model (including cross validation) ; 5min without cross validation
- **TF-IDF:** ~2h each supervised learning model (including cross-validation) ; 30min without cross validation
- **Word2Vec:** ~2h each supervised learning model
- **Pre-train Word2Vec**: Less than 1 min for each LSTM and bidirectional LSTM to process news titles; Less than 10 min for each LSTM and bidirectional LSTM to process news bodies

- **GloVe:** Less than 1 min for each LSTM and bidirectional LSTM for news titles; Less than 10 min for each LSTM and bidirectional LSTM to process news bodies
- **Doc2Vec:** ~2h each supervised learning model
- **FastText:** ~1min to process each part of the articles (the title and the body)

4.3 Model Selection

By comparing the performance results, the most performant algorithm in term of f1 score is Word2Vec trained on the data set (not the pretrained model). This algorithm outperforms any other algorithm on almost every metrics. But, it is very long to run as it take about 2h. If, in the future, we want to favor time, Fasttext is a good compromise as it gives very good performance, almost as good as for Word2Vec, so it is a good compromise.

4.4 Next Steps

For the next steps, we would like to perform the following tasks:
- Test our models on different articles (at least for the text data). - Think about how our model can have an added value, and who would be interested by a such model. - Reflection on how potential customer could use our model and how to integrate them in a process of fake news detection.

#Appendix

1. All materials and Colab notebooks are saved in this google drive path: https://drive.google.com/drive/folders/1a4KrqZ0GToEW3tm0mnz5OvAdkEymhGFG?usp=sharing
2. FastText sources: Enriching Word Vectors with Subword Information : https://arxiv.org/pdf/1607.04606.pdf - Bag of Tricks for Efficient Text Classification: https://arxiv.org/pdf/1607.01759.pdf
3. Understanding tf-idf: https://monkeylearn.com/blog/what-is-tf-idf/
4. How to use tf-idf: https://towardsdatascience.com/introduction-to-nlp-part-3-tf-idf-explained-cedb1fc1f7dc
5. GloVe implementation: https://github.com/stanfordnlp/GloVe
6. Train Word2Vec models: https://wiki.pathmind.com/word2vec
7. Text Classification with Doc2Vec: https://towardsdatascience.com/multi-class-text-classification-with-doc2vec-logistic-regression-9da9947b43f4