Fake News Detection using news content and user engagement

A dissertation submitted in fulfillment for the degree of
Master in Data Engineering and Science

by

Mario Pérez Madre

Director: Álvaro Rodrigo Yuste

Universidad Nacional de Educación a Distancia

June, 2021

Abstract

Fake news are purposefully designed to be misleading, and their success depends mostly on their readers [ÖG17]. Due to the nature of social networks, fake news can be quickly propagated, potentially causing a great damage to the society. Moreover, sociological phenomena like echo chambers or polarization [Sil+16], and psychological factors like confirmation bias [Del+16] or overconfidence to be fooled by fake news [1], create the perfect playground for misinformation [Tac+17] [Del+18]. Platforms have recently adopted measures [2] to discourage users sharing content without reading it first, but these measures are still not fully enforced and easy to bypass. An automatic fake news detection system that blocks or warns users about possibly misleading information will be needed in the near future, especially with the high volume of information that is shared through these websites.

Several models have been proposed to detect fake news by analyzing linguistic features [HA17] [Pot+17] [BS19] [KGN21], but these are often not enough [Shu+18] to distinguish fake news from real ones. Research is now focusing on including user engagement information to existing content-based models [RSL17] [Del+18] [SML19]. Some systems have been proposed that only use information from these engagements [Tac+17]. Part of the research has focused on creating training datasets [SW18] [Shu+18] for this task, crawling news from fact-checking websites and fetching user engagements using the public APIs offered by social networks.

In this work, we develop and test different fake news detection systems using information from news articles and user engagements in social networks. Two different architectures are used. The main one is based on Deep Learning, and can process news content and user engagements. The second one is based on well-known algorithms like logistic regression, SVM, random forest, LightGBM or XGBoost; it can only process news content and is used as a performance baseline for our Deep Learning models.

We use the FakeNewsNet dataset [Shu+18], which contains real and fake news from two fact-checking sources. For each news piece, this dataset contains the scraped news article, as well as tweets and retweets related to each news, and user profiles of the users involved in these tweet, including the user's timeline, followers and followees, although not all this information will be used.

Our work starts with an exploratory data analysis on the train set, where we highlight the main characteristics of the dataset. Then, we carry out a series of tests on both architectures, taking news from each set of news, with different subsets of features and with various textual representation techniques. Additionally, we perform an ablation test on the Deep Learning architecture, to understand how individual features behave and how do they complement each other.

Our results clearly show that our architecture is able to capture much information from user engagements, and that including user interactions gives better results than models using only information from news articles.

[1]`https://www.journalism.org/2016/12/15/many-americans-believe-fake-news-is-sowing-confusion/`
[2]`https://techcrunch.com/2020/09/24/twitter-read-before-retweet/`

With this work, our main contribution is a Deep Learning architecture capable of handling varying-length sequences of engagements for each piece of news, while also extracting all the information from them without padding or truncating to fixed-size sequences. We take advantage of recent innovations in frameworks like Tensorflow to process non-tabular-shaped data, which allows to directly include unaggregated features, minimizing the preprocessing required before the input data is fed to the model. This architecture can perform complex summarizations, such as a trainable recurrent layer that takes a sequence of user engagements in the same order as they were published, and outputs a vector that summarizes the whole user engagement sequence.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Fake news are news intentionally created to be misleading or deceptive. They are designed to spread misinformation, and their success depends almost entirely on their readers, who have the power of sharing them, or stopping them [ÖG17]. Social networks are currently the most powerful ally of fake news: an unsupervised, unregulated space where information can be globally, instantly accessed and people tend to spend only a few seconds on each post. They are a perfectly-tailored propagation media to spread misinformation [Tac+17] [Del+18]. Of course, due to their size, social networks will still be unsupervised in the near future, at least not by humans. Hence, the solutions are (1) educate users on how to distinguish fake news from real news, or (2) develop a fake news detection system that is able to block fake news or warn users about a possible lack of veracity.

According to a survey [1] conducted by Pew Research Center in the United States in December 2016 (after the U.S. election), 64% adults say fabricated news stories cause a great deal of confusion about the basic facts of current issues and events. Moreover, 39% feel very confident that they can recognize fabricated news and another 45% feel somewhat confident. The survey also reveals that 23% say they have ever shared a made-up news story, with 14% of respondents saying they shared a story *they knew was fake* and 16% having shared a story they later realized was fake. Another survey [2] conducted by Ipsos for BuzzFeed News a few days earlier revealed that 75% of respondents who recognized a fake news story from the U.S. election still viewed the story as somewhat or very accurate.

It is clear, then, that people overestimate their ability to distinguish fake news, and it is remarkable that some readers purposefully shared fake news. In fact, social networks boost the effect of echo cambers and group polarization, particularly in politics [Sil+16]: the more a user reads comments or news from a page, the more news they will receive from that page in the next days. Users are pushed further and further until they are polarized enough to share fake news, even if they know they are not true. In particular, this process ends up clustering users [RSL17] in groups where all participants share the same bias [Del+16].

Some measures have been taken recently to encourage users to read news articles before sharing them [3] [4] or to warn users about old articles that might be falsely spread as recent news [5]. However, these measures have a limited effect, since platforms do not force users to read the articles yet. Even if users had to read those articles, this measure would not be effective for polarized users [Sil+16], who would surely agree with the content of any news they share. Therefore, it is still necessary to implement some kind of fake news detection system.

[1] https://www.journalism.org/2016/12/15/many-americans-believe-fake-news-is-sowing-confusion/
[2] https://www.buzzfeednews.com/article/craigsilverman/fake-news-survey
[3] https://techcrunch.com/2020/09/24/twitter-read-before-retweet/
[4] https://techcrunch.com/2021/05/10/facebook-pop-up-read-before-you-share/
[5] https://www.theverge.com/21304173/facebook-old-news-articles-warning-notification-time

There exist several fact-checking agencies [6] in many countries, whose job is to read all news and judge whether they are true or not. However, their journalists have to perform the fact-checking manually, and given the increasing rate of published news, they can only analyze the most popular ones.

An automatic fake news detection system is needed. Several models have been proposed to detect fake news by analyzing linguistic features, not only using news articles and titles, but also the writing style, complexity, punctuation, capitalization and semantic meaning [HA17] [Pot+17] [BS19] [KGN21]. Although linguistic features like these contain information that differentiates fake and real news [HA17], they are often not enough to detect fake news [Shu+18]. Therefore, the next step is to include features about user engagement in social networks [RSL17] [Tac+17] [Del+18] [SML19], especially with surveys like mentioned above proving that a large fraction of users are unable to detect fake news and that there exist groups of people who consciously share fake news.

In this work, we develop and test different fake news detection systems using information from news articles and user engagements in social networks. Two different architectures are used. The main one is based on Deep Learning, and can process news content and user engagements. The second one is based on well-known algorithms like logistic regression, SVM, random forest, LightGBM or XGBoost; it can only process news content and is used as a performance baseline for our Deep Learning models.

We use the FakeNewsNet dataset [Shu+18], which contains real and fake news from two fact-checking sources. For each news piece, this dataset contains the scraped news article, as well as tweets and retweets related to each news, and user profiles of the users involved in these tweet, including the user's timeline, followers and followees, although not all this information will be used.

Our work starts with an exploratory data analysis on the train set, where we highlight the main characteristics of the dataset. Then, we carry out a series of tests on both architectures, taking news from each set of news, with different subsets of features and with various textual representation techniques. Additionally, we perform an ablation test on the Deep Learning architecture, to understand how individual features behave and how do they complement each other.

Our main contribution with this work is a Deep Learning architecture that is capable of handling varying-length sequences of engagements for each news piece, while also extracting all the information from them without padding or truncating to fixed-size sequences. Our approach is somewhat similar to [RSL17] and [SML19]. However, we take advantage of Tensorflow's Ragged Tensors [7], introduced a few years ago, to process non-tabular-shaped data, which allows to directly include unaggregated features, so that no preprocessing has to be done before the input data is fed to the system. While existing models propose to aggregate user engagements in temporal windows, our architecture can process individual engagements. Moreover, our architecture can include complex summarizations, such as a trainable recurrent layer that takes a sequence of user engagements in the same order as they were published, and outputs a vector that summarizes the whole user engagement sequence. This might be particularly interesting for early fake news detection.

[6] `https://en.wikipedia.org/wiki/List_of_fact-checking_websites`
[7] `https://www.tensorflow.org/guide/ragged_tensor`

1.1 Objectives

Our main goal is to check whether including user engagements in our models results in a performance improvement with respect to using only news articles, and what features contribute most to that improvement. We break this general goal into a set of specific goals:

- Study how are features correlated between them and with news labels.
- Create a Deep Learning fake news detection system that can use all the information available from news content and user engagements, and can handle varying-length sequences of user engagements without padding or truncating to a fixed length.
- Test fake news detection models using different textual representation techniques, including Vector Space Models and pretrained embeddings like Word2Vec or BERT.
- Study the performance impact of each feature depending on their type (numerical, categorical, textual) and origin (news, user engagement, user profile).

1.2 Document Structure

The structure of this document is as follows:

- We start in chapter 2 by reviewing the state-of-the-art fake news detection models, especially some of the few systems that include user engagements.
- In chapter 3, we explain the reasons why we decided to use the FakeNewsNet dataset and we carry out an exploratory data analysis and a study on the correlation between features and with news labels.
- We explain in chapter 4 the two architectures mentioned above, leaving the implementation details in appendix A.
- Next, we present all the experiments and analyze our results in chapter 5. We include there the main results (F1 scores on the test set), and leave the rest (accuracy, precision-recall AUC and ROC AUC) in appendix B.
- Finally, we gather the most relevant conclusions and mention possible future work in chapter 6.

Chapter 2

State of the Art

Traditional fake news detection models used to focus on news linguistic features. Many of them relied on news text representations using Vector Space Models or pretrained embeddings like Word2Vec or GloVe, although other linguistic features are commonly used, like writing style, complexity, punctuation, capitalization and semantic meaning [HA17] [Pot+17] [BS19] [KGN21]. However, due to the deceiving nature of fake news, linguistic features are sometimes unable [Shu+18] to detect fake news, and hence, research started shifting towards studying user behavior [RSL17] [Tac+17] [Del+18] [SML19]. Fake news are usually targeted [Del+16] to a specific public, and therefore, studying which, how and how much users react to news is key to understanding fake news propagation. The question, then, is how to use user information, and how to integrate it with news features.

Depending on whether models use news or user features, they can be classified [ÖG17] as content-based (textual and non-textual features from news), network-based (connections between users), user-based (user information) or hybrid. State-of-the-art models are in the hybrid class, although their advances come mainly from new user-based approaches, which are then merged with traditional content-based models. Therefore, advances in linguistic-related models are also important to the fake news detection task, and we must review the most recent innovations in textual representations.

Another important question is how and where to obtain training data from. The main issue is that most social networks currently sell their public data, and have paid subscriptions or small rate limits. Therefore, their Terms of Use are usually very strict, and do not allow to directly share any data. Instead, researchers willing to share a dataset usually rely on comment and user identifiers, and develop a set of tools that lets other people replicate the dataset. Although this approach allows the dataset to grow, data is subject to change over time, and most importantly, can become unavailable in the future, especially with recent privacy policies in areas like the European Union, which have made more difficult to obtain unaggregated data. As a consequence, datasets for this task are scarce and usually very small, with a few exceptions, and are subject to substantial change over time.

This chapter is divided in three sections. In section 2.1, we review the state-of-the-art fake news detection models, especially those which have successfully combined news and user features. We explore in 2.2 the most recent advances in textual representations, and finally, we discuss in section 2.3 the available datasets.

2.1 Fake News Detection Models

In this section, we review the most interesting and recent fake news detection models, especially the few that include user engagements. Most architectures use a small amount of information from user interactions. Namely, they apply a singular value decomposition to the news-user incidence matrix, like the CSI [RSL17] and SAF [SML19] models, and append some other information like the frequency and delay of engagements and some representation of the engagement content. There is still a lot of room for improvement regarding what features can be used for this task, with the additional difficulty of modeling the one-to-many news-engagements relationship, and it is even more difficult to find a suitable model for the underlying user network.

Capture-Score-Integrate In words of the authors of [RSL17], features can be divided in three groups: *text* (news content), *response* (user engagements) and *source* (news origin characteristics). They developed a new model called Capture-Score-Integrate (CSI), which they state is the first model using the three types of features, and showed that it performs significantly better than previous state-of-the-art models in datasets with user reactions from Twitter and Weibo. The model consists on a Capture module, which extracts information from the news text and user engagement (*text* and *response*), a Score module that evaluates the news target public (*source*), and the Integrate module, that integrates both modules and does the classification. These modules are built as follows:

- **Capture.** In essence, this module is just a LSTM recurrent layer whose input is a sequence of vectors containing information from news or user engagements in a temporal window. For each non-empty temporal window, its associated vector contains the number of engagements, the average delta time between engagements, the average user vector representation and the average textual representation of the engagements (or the article, if it is the first window). User representations are obtained from a binary incidence matrix between users and news (which users are related to which news), as a result of the singular value decomposition of that incidence matrix. Textual representations are obtained using *doc2vec* on each news article or engagement text. As we can see, although it is true that the model does use the *response* part, the information obtained from users is limited to an incidence matrix and the delay and frequency of engagements.

- **Score.** This module consists on two stacked fully connected layers, the first one with the hyperbolic tangent as activation function, and the second one using the sigmoid function. Firstly, a user graph is constructed by computing the number of times a pair of users comment in the same article, and then a vector representation for each user is obtained by computing the singular value decomposition of this matrix. The matrix of user representations is used as input to the Score module (note that this matrix does not depend on the news piece) and, in the end, a score between 0 and 1 is obtained for each user. Again, the information extracted from the *source* part is just the user engagement graph, ignoring all the features of each user.

- **Integrate.** This module glues the previous modules and computes the final prediction. The output of the Capture module goes through a fully connected layer with hyperbolic tangent activation, and the resulting vector is concatenated with the average score of the users who engaged with the given news piece. This vector, then, is passed to a final dense layer with sigmoid activation, which predicts the news label.

Notice how, even with such little information from the *response* (a binary news-user incidence matrix and the frequencies and delays of engagements) and *source* features (a user coincidence graph), the CSI model is able to outperform previous state-of-the-art models.

User-based Facebook Hoax Detector The authors of [Tac+17] propose a model that only uses the binary incidence matrix between posts and users who liked each post to predict whether each post is a hoax. The dataset, described in section 2.3, consists on posts from either science pages (considered as non-hoax) or conspiracy pages (considered as hoax), and the set of users who liked each post. Using only the incidence matrix of the training set, they train a logistic regression or a harmonic boolean label crowdsourcing algorithm, which, as the authors state, is commonly used to correct bias effects in polls, for instance.

In the logistic regression setting, the goal is to learn the weights ω_u associated to each user. On the other hand, the harmonic boolean label crowdsourcing algorithm described in the article is an iterative process. The core of the algorithm is to model the probability of a post i being non-hoax as the mean of a beta distribution with parameters α_i, β_i, this is, $p_i = \frac{\alpha_i}{\alpha_i+\beta_i}$. Similarly, the probability of a user u being truthful (to prefer non-hoax posts) is modeled as the mean of a beta distribution with parameters α_u, β_u. The parameters α are linked to the number of non-hoax posts liked by each user (or the number of truthful users that liked a post), while parameters β are connected to the number of hoax posts (or the number of untruthful users who liked them). The algorithm requires to know the ground truth of a small set of hoax and non-hoax posts, and assigns initial values α_u, β_u to each user, knowing which of them liked each starting post. Then, the algorithm updates the values of the parameters α_i, β_i based on the mean of the distribution of each user. These two steps are repeated until convergence is reached (the authors only needed five iterations).

The authors claim that the model achieves more than 99% accuracy on the test set, even when using 1% of training data. However, the dataset assumes that the posts from each page are all hoaxes or all non-hoaxes, which does not hold in practice. This assumption greatly simplifies the task, especially considering that users who like a page, usually like all of its posts indiscriminately. If those pages were assumed to contain both hoaxes and non-hoaxes, the model accuracy could drop significantly. In short, the authors of this article have only proved that, if a certain publisher is publishing only hoaxes or only non-hoaxes, then their model is capable of distinguishing which type of content is distributing. However, their results confirm that users tend to create groups where participants are more prone to like hoaxes.

Social Article Fusion Following the core principles of the CSI model [RSL17], the creators of the FakeNewsNet dataset [Shu+18] also defined a model [SML19] named Social Article Fusion (SAF) that uses news features and user engagements. As in the CSI model, SAF is divided in two modules that are joined to form the complete system:

- **News content (SAF /S).** This module follows an autoencoder structure, where the decoder and encoder are formed using recurrent layers (the article does not clarify which type of recurrent layers, though). News text is vectorized (presumably using a Vector Space Model) and passed to the encoder, which tries to learn a lower dimensional representation, and the decoder tries to reconstruct the original text. The last hidden state of the encoder is used later as the news features to the classifier head.

- **Social engagements (SAF /A).** As in the *Capture* module of CSI, the binary incidence matrix of news and user engagements is computed and decomposed by its singular values to obtain a user representation, which is joined with the textual content of the engagement, and presumably other features (the article does not mention explicitly all the features that are used here). The sequence of vectorized engagements is passed to a LSTM layer, and the last hidden state is passed to the classifier head, representing the information from social engagements.

- **Classifier head.** Finally, the latent news text representation and the social engagement vector are concatenated and passsed to connected to a fully connected layer, and the softmax activation is computed to obtain the final predictions.

To train the model, the authors set the model loss to be the sum of the loss of the autoencoder plus the loss of class label predictions, with an extra regularization term. Both losses are computed as the cross-entropy between the true word (or news label) and the predicted one.

The authors tested this model on the FakeNewsNet dataset in [Shu+18], using only SAF /A, SAF /S or the whole SAF, and compared them with other algorihtms like SVM, logistic regression, naive Bayes or CNN (all these using only news text). They showed that SAF achieves better results than SAF /S and the other content-based algorithms, and that using the user-based SAF /A model gives reasonable results. This means that user engagements actually give extra information. However, as we pointed in the CSI model [RSL17], it uses little information about the engagements (presumably, only the text, the SVD user vector and perhaps the number and delay of engagements).

Deep Diffusive Network This model, introduced in [Zha+18], exploits the relationship between news creators (the person who claims the statement contained in the article), news articles and news subjects (topics). They show that Deep Diffusive Network outperforms similar models that also depend on this relationship. It does not include user engagements, though, but its graph-based structure makes it rather unique for this task.

Each news article has a textual content and a credibility label. Similarly, each creator is given a description (a short sentence with the most relevant background or job) and a credibility label, and each subject is given a textual description and a credibility label. The article does not clarify how are assigned these two credibility labels.

The proposed model follows a graph structure: each article, creator or subject has a node, with each creator linked to one or more articles and each article connected to one or more subjects. Each node contains an inner model where inputs are fed to a Hybrid Feature Learning Unit (HFLU) layer, followed by a Gated Diffusive Unit (GDU) layer, which outputs a state vector. These custom layers are created as follows:

- **HFLU.** This layer takes as input the text string of a news article, creator or subject, and extracts features in two different ways. Firstly, a vocabulary (different for each type of node) is used to represent the given string with a bag-of-words Vector Space Model. Secondly, a RNN layer is used on the tokenized string and the final hidden state is obtained. Finally, both representations are concatenated and passed to the GDU.

- **GDU.** This layer is depicted in the article with a flow diagram, showing a structure similar to a LSTM cell. The inputs are the HFLU output and the output of other GDU layers connected to the current node. All the outputs from creator GDU layers are averaged, and separately, the outputs from subject GDU layers are also averaged. A GDU consists on four gates (*forget*, *adjust* and two *selection* gates). The first one can modify the subject averaged input, while the second changes the creator average vector. The first selection gate controls how much information from the new subject vector is introduced (and how much is discarded from the old vector). Similarly, the other selection gate controls the creator vector. The final hidden state vector is computed as the sum of the four open-close combinations of both selection gates.

Each pair of connected nodes output their states to the other node, as inputs to the GDU layer. From each node, the final hidden state goes through a dense layer (same weights within each type of node) and a prediction vector is computed with a softmax activation. The objective function is to minimize the sum of losses of the nodes (plus a regularization term), with each node loss being the cross-entropy between predicted and true labels, and the model is trained using backpropagation.

Hybrid Fake News Detector The authors of [Tac+17] created this model in [Del+18], which applies a user-based method if the news post (the post that links to the news article) has at least λ user likes, or a content-based method otherwise. They tested the model (with $\lambda = 3$) using the dataset from [Tac+17], and also the PolitiFact and BuzzFeed collections from FakeNewsNet [Shu+18], showing results comparable to [Tac+17] in its dataset (99.1% accuracy with 10% train size), and much better than other state-of-the-art models in both FakeNewsNet news sets, especially in the PolitiFact collection (92.1% F1 score). The architecture of content-based and user-based model is the following:

- **Content-based.** The content-based model starts by concatenating the text of the news post with the title and text preview of the link, and with the scraped news article (FakeNewsNet news are already scraped during the download process). Then, each word of the concatenated string is stemmed, the string is represented as a TF-IDF vector, and logistic regression is used to obtain a predicted label.
- **User-based.** The user-based model is exactly the same as in [Tac+17]. Two different machine learning algorithms are tested: logistic regression and harmonic boolean label crowdsourcing.

Note that only one of those models is used for each news piece. As an interesting novelty, the authors developed a Facebook bot that takes as input the URL of a Facebook post, applies the corresponding model depending on the number of likes, and returns a prediction.

2.2 Textual representations

BERT and BERT-based models like SentenceBERT are currently the state-of-the-art regarding textual representation, and have dethroned popular techniques like Word2Vec [Mik+13] or GloVe [PSM14], which had previously proved better than traditional Vector Space Models. We describe both models in the following paragraphs.

2.2.1 BERT

Bidirectional Encoder Representations from Transformers [Dev+18] was designed to be pretrained once and fine-tuned for each task. The architecture of BERT is a chain of Transformer models [Vas+17]. Each Transformer model follows an encoder-decoder style, built using multi-head attention layers (also called *self-attention* layers) and fully connected layers. Multi-head attention layers are a concatenation of multiple attention layers. The encoder consists on a multi-head attention layer and a fully connected layer. The decoder receives the encoder output and the output of a masked multi-head attention layer which reads the outputs of previous positions. Then, the decoder continues with a non-masked multi-head attention layer and a fully connected layer. It is worth mentioning that the encoder and decoder inputs go through an initial embedding, which also uses positional encoding, and that the result of each multi-head attention and fully connected layer is always summed with their input, and then, normalized.

The authors tried different BERT sizes, changing the number L of Transformer blocks, the size H of multi-head attention and fully connected layers, and the number A of heads in multihead layers. The main BERT model is BERT-base, which has $L = 12$, $H = 768$ and $A = 12$. Therefore, BERT base consists on 12 Transformer blocks, where fully connected and multi-head attention layers have 768 hidden cells and the attention layers concatenate the output of 12 attention layers of size 64 each.

BERT is pretrained on unlabeled data, trying to optimize for two tasks at the same time:

- **Masked Language Modeling.** Some tokens are masked and BERT tries to guess the tokens.
- **Next Sentence Prediction.** BERT has to decide if the second sentence immediately follows the first.

It accepts either one or two input sequences, so it can be directly used for tasks like Question Answering. The first token is always a `[CLS]` token. Then come the tokens from the first sentence. If two sentences are passed, a special `[SEP]` token is used to separate them, and finally the tokens of the second sentence are added. Masked words are tokenized as `[MASK]`.

The ultimate goal is to obtain a representation for each token. These representations are the last hidden states corresponding to each token. In particular, the `[CLS]` token representation is used in the NSP task and can interpreted as a sentence-level representation, although the authors state that without fine-tuning, it is not a meaningful representation of the sentence, since it is trained for the NSP task.

BERT has two main advantages:

- **Deeply bidirectional context.** Previous models used a left-to-right approach. Some of them concatenated left-to-right and right-to-left representations, but were not able to interact between them. However, BERT can use a bidirectional context, since the Masked Language Modeling task is not trivial when both sides of the word are known, unlike other tasks. The authors state that not using bidirectional context might harm performance for token-level tasks like Question Answering.
- **Minimal changes to adapt for a specific task.** Once BERT is pretrained, it is only needed to fine-tune the model for the task at hand.

2.2.2 SentenceBERT

As we mentioned above, BERT's `[CLS]` token could be used as a sentence-level embedding, after fine-tuning is performed. Other researchers tried to use the average token embedding as sentence embeddings. SentenceBERT [RG19] is a modification of the original BERT model to obtain sentence representations that are semantically meaningful with respect to the cosine similarity, this is, such that similar sentences obtain similar representations. The authors carried out tests using the `[CLS]` token, the average of token embeddings and the max pooling of token representations, showing that SentenceBERT obtains much better results. It is surprising that, in some cases, taking the average token embedding or the `[CLS]` token from the original BERT model performed worse than taking the average GloVe embeddings (see table 5 in [RG19]). Finally, they show that SentenceBERT performs better taking the mean pooling or the `[CLS]` token embedding than the maximum pooling.

2.3 Datasets

In this section, we explore the most relevant datasets in the fake news detection context. As we explained earlier, one of the problems with datasets that contain social information is that platforms usually do not allow to directly share the dataset. Instead, the authors can only share tools to download the dataset, which allows the dataset to grow over time, but has a major disadvantage: when platforms change their Terms of Use, and it happens quite frequently, parts of the dataset or even the entire dataset might be inaccessible from that moment. For instance, Freedom's Daily, one of the nine Facebook pages from the BuzzFace dataset, is no longer accessible. Platforms might also limit download speed, especially those which have paid subscriptions that allow higher rate limits. These are some reasons why these datasets were so scarce and small until recent years.

Let us describe the most relevant datasets available:

2.3.1 CREDBANK

The CREDBANK dataset [MG15] was built from tweets collected between October 2014 and February 2015. The authors tracked more than 1 billion tweets (1% of all tweets) into a real-time topic-modeling system based on Latent Dirichlet Allocation, grouping the tweets into categories (using the top three topics from LDA) that may or may not represent an event. A group of people was asked to manually label these groups as *events* or *non-events*. Then, these annotators were asked to assess the credibility of these events, giving them a rating from -2 (*certainly inaccurate*) to +2 (*certainly accurate*). In total, 1.049 categories were labeled as *events*. A final Twitter search was carried out to find more tweets related to these *events*.

In the repository description, the authors state that the dataset contains 169 million streaming tweets, grouped in more than 1.300 labeled events, and another 80 million tweets from the final search. Therefore, we assume that these are only the tweets categorized as events, and that the public dataset is a bit larger than the dataset described in the article, since it contains more events.

2.3.2 BuzzFeedNews

BuzzFeed journalists [Sil+16] analyzed 2.282 Facebook posts fed from 9 known political Facebook pages (3 left-biased, 3 right-biased and 3 mainstream) from September 19-23, 26 and 27, 2016 (six weeks before the 2016 U.S. elections). Some posts contained links to the news article, while some others were videos. They fact-checked every post and classified them as *mostly true*, *mostly false*, *mixture of true and false* or *no factual content* if it had no factual claims.

The article shows that all left-biased and right-biased pages published a high number of mostly false or partly false posts, while mainstream sources only published a few partly false posts, the majority of them related to the same story. Regarding user engagement, posts that were mostly false or partly false were shared much more frequently than mostly true posts. An important remark is that many posts categorized as *no factual content*, such as pictures or videos, were politically biased and generated high user engagement. Another interesting point is that right-biased posts almost never contained references to mainstream sources (they mention that, even when news came from a mainstream page, they usually linked to right-biased sites), whereas left-biased posts often contained links to mainstream pages. As the article explains, these crossed references, combined with the feed algorithms used by Facebook or Google (which are based on previous searches), generate a cycle that results in group polarization.

The dataset gathered by Buzzfeed journalists contains, for each post, the Facebook ID of the post and the poster, the rating given to the post and information about user engagements (shares, reactions, comments), although the posts themselves can be downloaded using the Facebook Graph API.

2.3.3 Some Like It Hoax

This dataset [Tac+17] consists on all the Facebook posts fed from 18 selected scientific Facebook pages (whose posts are considered *non-hoax*) and another 14 conspiracy Facebook pages (categorized as *hoax*) from July 1st, 2016 to December 31, 2016. At the time it was published, the dataset contained 15.500 posts involving more than 900.000 users. The authors provide a set of scripts to download the dataset using Facebook's Graph API. This dataset was tested in [Tac+17] using only information from the users who liked each post, showing extremely good results even when training with a tiny portion of users.

2.3.4 BuzzFace

Published in 2018, this dataset [SW18] is an extension of BuzzFeedNews containing all the replies from the original Facebook posts. The authors provide a set of scripts to download the trees of Facebook posts (with all metadata) that arise from the original fact-checked posts, and they also provide tools to download and scrape the news articles contained in the original posts. By trees we mean the entire discussion threads starting with each post, not only the direct replies to the post. In total, there are 1.7 million posts. If the news site has Disqus comments or embedded tweets, they can also be downloaded.

At the time this dataset was published, it was the most extensive dataset containing social information from Facebook. Moreover, as the authors state, they multiplied by 400 the number of Facebook comments contained by the previous state-of-the-art dataset in news veracity assessment. They also mention that the only relevant datasets that they found contained Twitter posts. One of them could be the CREDBANK dataset, mentioned above.

2.3.5 FakeNewsNet

This dataset [Shu+18] contains real and fake news from two fact-checking sources: *PolitiFact* (948 news with text) and *GossipCop* [1] (21.714 news with text). For each piece of news, the dataset contains the scraped news article, as well as tweets and retweets related to each news, and user profiles of the users involved in these tweet, including the user's timeline, followers and followees. Many news have associated images and videos, and some news contain only non-textual content. Almost two million tweets are related to the included news, and more than a billion users form the complete follower network, with around half a million users that posted at least one tweet. Moreover, the dataset contains spatiotemporal information about many tweets, users and news. The authors provide some scripts to download the collection, with the possibility of downloading specific parts of it (for example, only news and tweets or only data from PolitiFact). The total size of the dataset is huge, and it would take a long time to download all of it.

[1]The authors clarify that fake gossip news are extracted from *GossipCop*, but real news are extracted from *E! Online*.

Chapter 3

FakeNewsNet Dataset and Analysis

We saw in chapter 2 that including information from user engagements in content-based models has proved useful for the fake news detection task. After all, fake news are usually targeted to a specific public and the ultimate goal is to reach as much people and generate as many engagements as possible. Knowing information about the users that comment those news can give insights both on the target public of the news and on whether bot accounts are being used.

From now on, we will focus on the *FakeNewsNet* dataset. We will explain the reasons that motivated us to use this dataset in section 3.1. Next, we will cover, in section 3.2, the structure of the dataset and the resulting files. Thirdly, we will explain in 3.3 the process we followed to read the dataset and the extracted features. We will also discuss in 3.4 the cleaning and feature engineering process that we followed to mitigate some problems that came up as we explored the dataset. Then, in section 3.5, we will summarize the available features and split the data into separate train, validation and test sets. Next, we will carry out in section 3.6 an explorative data analysis on the train set, and finally, we will analyze the correlation between features, especially with class labels, in section 3.7.

3.1 Why FakeNewsNet?

We described in section 2.1 the available datasets that we found and were suitable to study the fake news detection problem. **We finally decided to use the FakeNewsNet dataset**, due to the following reasons:

- Since one of the goals is to compare the performance when social data is used, we need to have the tweets or posts related to each news, so we discarded the *BuzzFeedNews* dataset.
- Facebook's Graph API rate limits are much more strict than Twitter API rate limits. For instance, Facebook allows to lookup 200 posts per hour and per user (we would count as one user when downloading the datasets), while Twitter allows to lookup 900 tweets per 15 minutes and user, and we can easily multiply this number by using up to 10 keys with the same Twitter developer account. Therefore, we decided to avoid using the *BuzzFace* dataset (it would take us 8.500 hours to download the entire dataset) and the *Some Like It Hoax* dataset (it would take 75 hours to download the 15.500 posts but another 5.000 hours if we wanted the user profiles).
- We decided not to use the *CREDBANK* dataset because the news associated with the events are not directly available (we would have to manually review each event to find the associated news), and because it would take a very long time to download the entire dataset with the current rate limits (more than 2.000 hours to download just the 80 million tweets from the final search).

Moreover, the fact that FakeNewsNet has news from two different sources will allow us to compare model performances between both sources, so it is like having two datasets in one. Another nice feature is that the news scraping algorithms shared by the dataset authors also try to fetch the original news article from the *archive* and the *wayback machine*, in case the original website is not available.

3.2 Dataset structure and features

We followed the instructions given in the GitHub repository (see [Shu+18]) to download the dataset. The download process took more than 2 days to complete, spending most of the time downloading Twitter data. During the process, we requested multiple Twitter API access tokens to speed up the download.

Note. During the download process, we found that downloading the retweets, followers and followees parts of the dataset would be impossible. With a limit of 900 tweets per 15 minutes and key, downloading the retweets would take around 8.333 hours with 10 API keys, and the followers and followees have a much more strict limit of 15 users per minute and key. Therefore, **we decided to only use the news, tweets, user profiles and user timelines for both FakeNewsNet sources**. Unrelated to these rate limits, **we also decided not to use the pictures and movies contained in the articles and tweets,** although we will extract some information from the URLs. However, even with these restrictions, there is plenty of information to train fake news detection models.

The dataset follows a folder structure similar to figure 3.1. The upper levels are organized by source and label. Under each group, news are organized in distinct folders, each containing a file named `news content.json`, that contains the scraped news article, and the `tweets` folder, under which are stored the related tweets in separate JSON files. User profiles and user timelines are stored in the top-level `user_profiles` and `user_timeline_tweets` folders, respectively. Twitter-related files are named with the identifier of the tweet or user.

Twitter-related files revolve around the Tweet [1] object model and the User [2] object model. Note that the Tweet object contains a User object with information about the user that posted the tweet. Furthermore, if a tweet quoted another previous tweet, the quoted Tweet object will be contained contain in the replying Tweet.

To sum up, we have four different types of files:

- News files containing scraped fields from the original news articles.
- Tweet files containing information about each Tweet.
- User profiles files containing information about each User.
- User timeline files containing the most recent Tweets of the user, up to 200 Tweets.

It is important to be aware that the fields recorded in each type of file are not the same for all news. For instance, some news have metadata fields that are not present in other news, and some might contain incorrectly scraped fields. In the case of Twitter files, apart from the fact that many tweets do not quote previous tweets, some fields like `possibly_sensitive` or `has_extended_profile` are not present in all tweets or users, respectively, possibly because they were recently added to the Tweet object, while some fields referring to the user's status seem to be deprecated and might not be available in recent user profiles.

[1]https://developer.twitter.com/en/docs/twitter-api/v1/data-dictionary/object-model/tweet
[2]https://developer.twitter.com/en/docs/twitter-api/v1/data-dictionary/object-model/user

```
├── gossipcop
│   ├── fake
│   │   ├── gossipcop-1
│   │   │   ├── news content.json
│   │   │   └── tweets
│   │   │       ├── 886941526458347521.json
│   │   │       ├── 887096424105627648.json
│   │   │       └── ....
│   │   └── ....
│   └── real
│       ├── gossipcop-2
│       │   ├── news content.json
│       │   └── tweets
│       └── ....
├── politifact
│   ├── fake
│   │   ├── politifact-1
│   │   │   ├── news content.json
│   │   │   └── tweets
│   │   └── ....
│   │
│   └── real
│       ├── politifact-2
│       │   ├── news content.json
│       │   └── tweets
│       └── ....
├── user_profiles
│   ├── 374136824.json
│   ├── 937649414600101889.json
│   └── ....
└── user_timeline_tweets
    ├── 374136824.json
    ├── 937649414600101889.json
    └── ....
```

Figure 3.1: FakeNewsNet folder structure.

3.3 Reading and feature extraction

For each type of file (news, tweets, user profiles and user timelines) we will proceed as follows:

1. Iterate over all files to check which fields are present in all files. For the sake of simplicity, we will discard the fields that are not present in all files, although we could keep these fields and deal with empty values in the model's preprocessing pipeline.

2. Manually check if the remaining fields contain duplicated information, need some feature extraction technique or have a constant value.

3. Read the files applying the necessary feature extraction and save the information to disk.

In the following subsections, we will explain, for each type of file, which fields are available in all files, which are not available in some files and which fields will be retained. Our goal is to have a set of scalar fields for each type file, so we can store the information in separate tables, with each news being connected to a number of tweets and each tweet related to a Twitter user. News and tweets can be joined using the news identifier given by the dataset, while tweets and users can be joined by the user identifier, which is also contained in the Tweet objects.

Note. Some fields are not scalars, but dictionaries or lists. Whenever we come across a dictionary, we will drop it and keep only its inner fields. We will name those inner fields by concatenating the dictionary name and the field name with double underscores. Lists will be treated differently in each case, sometimes concatenating the values and sometimes counting the number of elements or taking the most frequent value.

3.3.1 News

We list, in table 3.1, the fields that were missing in at least one file and, in table 3.2, those that were found in all news files. All the missing fields are related to metadata, mainly referring to Facebook and Twitter. Note that the fields in table 3.2 already contain some metadata information, like `publish_date`. Overall, we consider that we are keeping most of the available information and dropping the missing fields will not have a big impact, since we will use tweets and users' information from Twitter data.

Fields
`meta_data__language`
`meta_data__news_keywords`
`meta_data__robots`
`meta_data__og__image__identifier`
`meta_data__og__image__width`
`meta_data__og__image__height`
`meta_data__twitter__url`
`meta_data__fb__admins`
`meta_data__fb__pages`
`meta_data__msvalidate.01`
`meta_data__viewport`
`meta_data__og__description`
`meta_data__fb`
`meta_data__description`
`meta_data__og__site_name`
`meta_data__og__type`
`meta_data__og__url`
`meta_data__og__title`
`meta_data__twitter`
`meta_data__twitter__title`
`meta_data__twitter__description`
`meta_data__twitter__image`
`meta_data__og__image`
`meta_data__og`

Table 3.1: News fields not present in all files.

Following the remarks in table 3.2, we dropped the fields `top_img`, `keywords`, `canonical_link`, `meta_data`, `source` and `summary`, since they contain no information or the information they give is already in another field. In the case of `top_img`, the reason is that we are not using the images.

To sum up, we keep the fields in table 3.3, applying the preprocessing specified to the right of each field. Note that we have added three more fields that will be extracted from the absolute paths: `news` (news identifier), `label` (*real* or *fake*) and `source` (*GossipCop* or *PolitiFact*).

Fields	Remarks
`url`	-
`text`	Some news contain only images or videos and others are not available
`images`	List of article's images URLs
`top_img`	Main image's URL. Many images are not available
`keywords`	Always empty
`authors`	List of sentences describing the author or media
`canonical_link`	Either coincides with `url` or is empty or less useful than `url`
`meta_data`	Dictionary containing metadata fields
`title`	-
`movies`	List of embedded videos' URLs
`publish_date`	Timestamp of the publish date
`source`	`url` up to the suffix part
`summary`	Always empty

Table 3.2: News fields present in all files.

Fields	Feature extraction
`url`	Get the domain, subdomain and suffix
`text`	-
`images`	Count and get the most frequent domain, subdomain and suffix
`authors`	Concatenate all sentences with whitespaces
`title`	-
`movies`	Count and get the most frequent domain, subdomain and suffix
`publish_date`	Convert timestamp to datetime
`news`	Extract from absolute path
`label`	Extract from absolute path
`source`	Extract from absolute path

Table 3.3: News fields and their feature extraction.

3.3.2 Tweets

We show, in table 3.4, the fields that were missing in some files and, in table 3.5, those that were found in all news files. Most fields refer to the quoted tweet (not all tweets are replies of previous tweets). The `extended_entities` fields are only used when media is shared natively inside Twitter, not with a link. The `withheld` fields are used when a tweet has been withheld due to a DMCA complaint, when the tweet violates any copyright. The `scopes` fields are used only for Twitter's Promoted Products to indicate who should receive the tweet. The field `possibly_sensitive` is used to indicate whether a link in the tweet might contain sensitive media. Finally, the field `has_extended_profile` indicates whether the user's status contains media shared natively (using Extended entity objects), although this field is currently deprecated.

Finally, we finally the fields shown in table 3.6, with the specified feature extraction. Following the remarks in table 3.5, we drop the fields as explained below. Note that, as with the news, we add a `news` field, with the news identifier for each tweet, so that we can join news and tweets later on.

- We drop all the `user` fields except `user__id`. We will read the user profiles from their JSON files.
- Next, we drop the fields `entities` (it is a dictionary, we will use the inner fields), `contributors`, `favorited` and `retweeted` (all values are empty), as well as the identifiers in string format: `id_str`, `in_reply_to_status_id_str` and `in_reply_to_user_id_str`. We do not need the `in_reply_to_screen_name` and `in_reply_to_status_id` fields either, because we will only use them to know whether the tweet is a reply or not.
- We drop the fields `geo` and `coordinates`, since `Place` contains the same or more information.

Fields
extended_entities
extended_entities__media
quoted_status_id
quoted_status_id_str
quoted_status
quoted_status__created_at
quoted_status__id
⋮
quoted_status__lang
withheld_scope
withheld_copyright
withheld_in_countries
scopes
scopes__place_ids
user__entities__url
user__entities__url__urls
user__profile_banner_url
possibly_sensitive
user__has_extended_profile

Table 3.4: Tweet fields not present in all files.

Fields	Remarks
created_at	-
id	Tweet ID as a 64-bit integer
id_str	Tweet ID as a string
text	-
truncated	-
entities	Dictionary to describe entity objects
entities__hashtags	List of Hashtag objects
entities__symbols	List of Symbol objects
entities__user_mentions	List of User Mention objects
entities__urls	List of URL objects
source	Utility to post the tweet
in_reply_to_status_id	Empty unless the tweet is a reply
in_reply_to_status_id_str	Empty unless the tweet is a reply
in_reply_to_user_id	Empty unless the tweet is a reply
in_reply_to_user_id_str	Empty unless the tweet is a reply
in_reply_to_screen_name	Empty unless the tweet is a reply
user	Dictionary holding user data
user__id	User ID as a 64-bit integer
user__id_str	User ID as a string
⋮	⋮
user__translator_type	-
geo	Coordinates from where the tweet was posted
coordinates	Coordinates from where the tweet was posted
place	Place object chosen by the user as its location
contributors	Always empty
is_quote_status	-
retweet_count	-
favorite_count	-
favorited	Always empty
retweeted	Always empty
lang	-

Table 3.5: Tweet fields present in all files.

Fields	**Feature extraction**
`created_at`	-
`id`	-
`text`	-
`truncated`	-
`entities__hashtags`	Concatenate text with whitespaces
`entities__symbols`	Concatenate text with whitespaces
`entities__user_mentions`	Count
`entities__urls`	Count and get the most frequent domain, subdomain and suffix
`source`	Get the source name using regular expressions
`in_reply_to_user_id`	Check if the tweet is a reply
`user__id`	-
`place`	Get the country
`is_quote_status`	-
`retweet_count`	-
`favorite_count`	-
`lang`	-
`news`	Extract from absolute path

Table 3.6: Tweet fields and their feature extraction

3.3.3 User profiles

Table 3.7 shows the fields that were missing in some files, and table 3.8 contains the fields found in all user profiles. Most fields refer to the `status`, which is the tweet users might choose to put in their user profile header. All these fields are similar to those of a Tweet object. Other missing fields are optional profile customizations like `entities__url__urls` (related to `entities__description__urls`), `profile_banner_url` (URL pointing to the user's banner image), `withheld_in_countries` (if the user profile is under a DMCA complaint), `statuses_count` (number of tweets published by the user) or `has_extended_profile` (related to media shared natively into their profile, similar to the `extended` Tweet fields).

Fields
`withheld_in_countries`
`entities__url`
`entities__url__urls`
`profile_banner_url`
`statuses_count`
`status`
`status__created_at`
`status__id`
⋮
`status__lang`
`has_extended_profile`

Table 3.7: User fields not present in all files.

We drop missing fields, and some other fields as per the remarks in table 3.8:

- Fields that contain the same or less information than others: `id_str`, `url`, `profile_location`, `profile_background_image_url_https`, and `profile_image_url_https`.
- Fields that are always empty or constant: `utc_offset`, `time_zone`, `lang`, `contributors_enabled`, `following`, `follow_request_sent`, `notifications`.

- We drop the `entites` and `entites__description__url` since we are keeping the inner field `entites__description__urls`.

The feature extraction process in this case is much shorter, since most of the fields store scalar values. The only feature extraction we need is to get the most frequent domain, subdomain and suffix from `entities__description__urls`.

Fields	**Remarks**
`id`	User ID as a 64-bit integer
`id_str`	User ID as a string
`name`	User name
`screen_name`	User alias, they are unique but may change
`location`	User location
`profile_location`	Either coincides with `location` or is empty
`description`	-
`url`	URL provided by the user, within Twitter's domain
`entities`	Dictionary
`entities__description`	Dictionary
`entities__description__urls`	Original URLs provided by the user
`protected`	If the user has chosen to protect their tweets
`followers_count`	If the account is verified (blue tick)
`friends_count`	-
`listed_count`	-
`created_at`	-
`favourites_count`	-
`utc_offset`	Always empty
`time_zone`	Always empty
`geo_enabled`	-
`verified`	-
`lang`	Always empty
`contributors_enabled`	Always false
`is_translator`	-
`is_translation_enabled`	-
`profile_background_color`	-
`profile_background_image_url`	-
`profile_background_image_url_https`	Like the previous but with HTTPS
`profile_background_tile`	-
`profile_image_url`	-
`profile_image_url_https`	Like the previous but with HTTPS
`profile_link_color`	-
`profile_sidebar_border_color`	-
`profile_sidebar_fill_color`	-
`profile_text_color`	-
`profile_use_background_image`	-
`default_profile`	If the user has not change the theme or background
`default_profile_image`	If the user has not uploaded a profile image
`following`	Always false
`follow_request_sent`	Always false
`notifications`	Always false
`translator_type`	-

Table 3.8: User fields present in all files.

3.3.4 User timelines

We found that Twitter user timelines are limited to the 200 most recent tweets. It is likely that the last 200 tweets of each user are not related to the news in the dataset, since the news are mostly from 2016 to 2019. Therefore, we decided to only count the number of tweets for each user's timeline and take that number as a measure of how active the user has been. Real users should have exactly 200 tweets whereas bots might have less than 200 tweets, since each one might be used to post only a few tweets. Therefore, **we restrict the user timeline information to this measure of activity and include it as another feature of user profiles.**

3.4 Cleaning and feature engineering

After reading the data and extracting all the useful features, we continued cleaning the dataset a bit more and crafting some new features. We came across some issues with some news being unavailable, for instance, and here we will explain how we dealt with these issues.

Cleaning

- Firstly, **we keep only news that have both text and title.** On the one hand, the dataset purposedly contains news with pictures or movies only, but we are focusing on the text, so we decide to remove these news. On the other hand, some news are unavailable or incorrectly scraped, with an empty `text` field and usually a generic sentence describing the media in the `title` field. Furthermore, some news are unreachable because of cookie pop-ups. These news have an empty `title` field and a cookie warning message in the `text` field, containing the usual message asking to accept the website's cookies. We did not notice any of these problematic news having both fields filled in, so we decided to remove any news with an empty `text` or `title` field.

- Next, **we remove any tweet that is not related to the remaining news, and all the tweets that were published more than a day before the piece of news they are related to.** All the tweets have an UTC datetime and many news have a publish datetime (without a specified timezone). We noticed that the publish datetime of some news was posterior to the post times of some tweets, which should not be possible, so either the publish date is wrongly scraped or the news was updated. Therefore, we decided that these tweets should be removed, so that all the tweets are related to the possibly updated version of the news. However, many tweets were published only a few hours before the news, due to their timezone not being the UTC timezone. We finally decided to leave a 1-day margin for this special case. Tweets related to news without a publish date were not removed either.

- Thirdly, **we remove the user profiles of users that do not own any of the remaining tweets,** as we are only interested in the users that posted at least one tweet.

Feature engineering

- Firstly, we compute, **for each piece of news, the number of tweets related to it.** It is reasonable to think that fake news have more tweets related to them.

- Secondly, we compute, **for each tweet, the time difference between this tweet and the previous tweet** of the same news. If there is no previous tweet related to the same news, we compute the difference between the tweet posting date and the news publish date. We should expect that fake news have smaller time differences, as the content should be more engaging.

- Finally, we compute, **for each tweet, the time difference between the tweet posting date and the user creation date.** This could be an indicator on whether the user is a bot account.

3.5 Final fields. Train, validation and test sets

At this point, we have a clean dataset containing the fields specified in tables 3.9, 3.10 and 3.11. Index columns are not shown in these tables: news are indexed by the `news` identifier extracted from the absolute paths, while tweets and users are indexed by their `id` and `user__id`, respectively. Note that we have added a prefix to each field to indicate whether the feature refers to news, tweets or user profiles. It will be quite helpful later on when using all the features, since many of them have similar names.

Fields	Types
`news__text`	string
`news__title`	string
`news__source`	category
`label`	int32
`news__num_images`	float32
`news__num_movies`	float32
`news__publish_date_datetime`	float32
`news__authors_text`	string
`news__url__subdomain`	category
`news__url__domain`	category
`news__url__suffix`	category
`news__images__subdomain`	category
`news__images__domain`	category
`news__images__suffix`	category
`news__movies__subdomain`	category
`news__movies__domain`	category
`news__movies__suffix`	category
`news__num_tweets`	float32

Table 3.9: News final fields.

Fields	Types
`tweet__created_at`	float32
`tweet__text`	string
`tweet__truncated`	category
`user__id`	int64
`tweet__is_quote_status`	category
`tweet__retweet_count`	float32
`tweet__favorite_count`	float32
`tweet__lang`	category
`tweet__news`	category
`tweet__entities__urls__subdomain`	category
`tweet__entities__urls__domain`	category
`tweet__entities__urls__suffix`	category
`tweet__is_reply`	category
`tweet__num_entities__user_mentions`	float32
`tweet__num_entities__urls`	float32
`tweet__country`	category
`tweet__source_name`	category
`tweet__entities__hashtags__text`	category
`tweet__entities__symbols__text`	category
`user__time_user_created_to_tweet`	float32
`tweet__time_delta`	float32

Table 3.10: Tweet final fields.

Fields	**Types**
`user__name`	category
`user__screen_name`	string
`user__location`	category
`user__description`	string
`user__protected`	category
`user__followers_count`	float32
`user__friends_count`	float32
`user__listed_count`	float32
`user__created_at`	float32
`user__favourites_count`	float32
`user__geo_enabled`	category
`user__verified`	category
`user__is_translator`	category
`user__is_translation_enabled`	category
`user__profile_background_color`	category
`user__profile_background_image_url`	category
`user__profile_background_tile`	category
`user__profile_link_color`	category
`user__profile_sidebar_border_color`	category
`user__profile_sidebar_fill_color`	category
`user__profile_text_color`	category
`user__profile_use_background_image`	category
`user__default_profile`	category
`user__default_profile_image`	category
`user__translator_type`	category
`user__entities__description__urls__subdomain`	category
`user__entities__description__urls__domain`	category
`user__entities__description__urls__suffix`	category
`user__timeline_length`	float32

Table 3.11: User final fields.

Note. **Not all these fields will be used as features.** For instance, news, tweet and user identifiers are only kept to be able to join them later on. Absolute datetimes like `news__publish_date_datetime` will not be used since, as we will see in 3.6, they introduce a bias that could make models perform better just by looking at the publish date, therefore exploiting a weakness of the dataset. We will later specify which features are excluded for training.

As a preparation for chapters 4 and 5, **we have standardized the internal types of each field.** Each field belongs to one of these groups:

- **Numerical.** We store numerical fields as 32-bit floating-point numbers, except the user and tweet identifiers (which are the user and tweet index fields, respectively), which are stored as 64-bit integers. Datetimes and timestamps are converted to 32-bit floating-point numbers and considered numerical features.
- **Categorical.** Categorical fields are stored as strings, replacing missing values by empty strings.
- **Textual.** Textual fields are stored as strings, except `tweet__entities__symbols__text` and `tweet__entities__hashtags__text`. These two features were created as concatenation of the text values they contained, but the former contains only 22 different values while the latter is mostly empty and 95% of the values correspond to 29 categories, so they are considered as categorical features.
- The `label` column, which contains the class label, will be stored as a 32-bit integer, using 0 for real news and 1 for fake news.

Finally, **we split the dataset into a train set (80% news), validation set (10% news) and test set (10% news)**, by randomly shuffling and splitting the set of news, stratified by source (*GossipCop* or *PolitiFact*) and label (*real* or *fake*). The reasons for those decisions are the following:

- We chose the 80-10-10 proportions because there is a lot of information, so we can afford to have separate validation and test sets with moderate size.
- News are shuffled because there is no need to make a chronological split. We assume that the underlying properties of real and fake news and their tweets have not changed during the period covered by the dataset. Therefore, training a model on newer news should not make it perform better in older news.
- We stratify by label to have a similar class distribution in all the sets. The decision to stratify by source is technical, since it allows us to keep a single set of train, validation and test sets. Instead of having two separate sets for each source and split, we can have one set and filter by source to obtain the corresponding set for that source.

We show in table 3.12 the number of news by source, split and class label. We see that the stratification proportions are correct.

		real	fake
source	split		
GossipCop	train	11898	3799
	val	1488	475
	test	1487	475
PolitiFact	train	349	299
	val	43	38
	test	44	37

Table 3.12: News by source, split and class label.

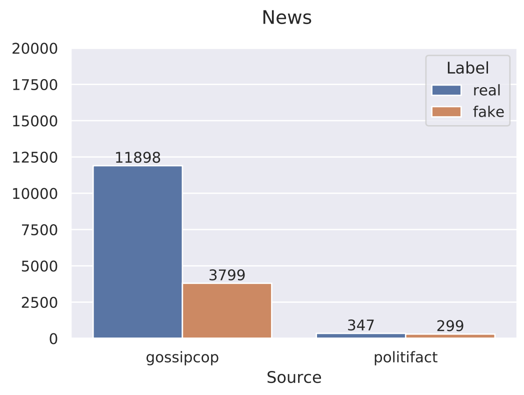
3.6 Exploratory data analysis

In this section, we carry out an exploratory data analysis on the training set, to familiarize ourselves with the data and to look for class biases and other possible issues we might have to deal with later on.

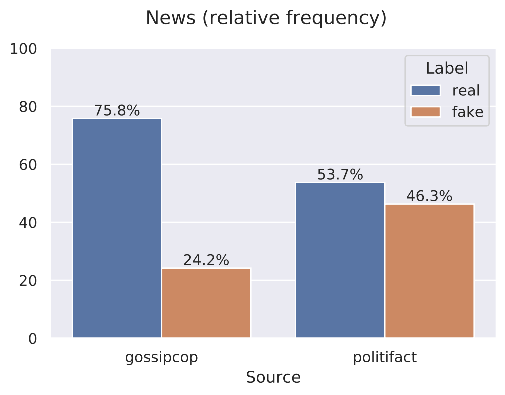
3.6.1 News features

We start by counting the number of news for each source and label. As we can see in figure 3.2a, we have many more news from *GossipCop* than from *PolitiFact*, and figure 3.2b shows that **GossipCop news are strongly unbalanced, with almost 76% of them being real news, while the PolitiFact collection is much more balanced.** We will have to keep in mind this bias.

However, there are many news with no related tweets. As we can see in figure 3.3a, 3.236 GossipCop news have no associated tweets, and 167 PolitiFact news are in the same situation. Note that the class distribution changes noticeably in the PolitiFact collection.

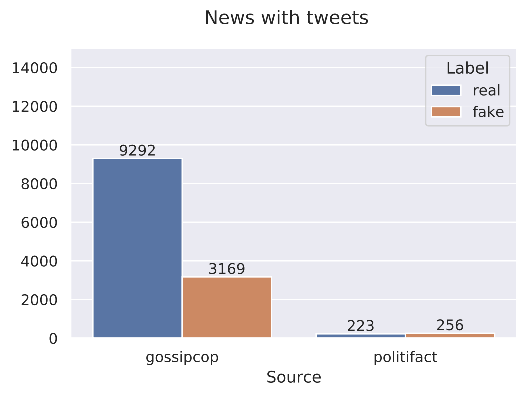


(a) Total count.

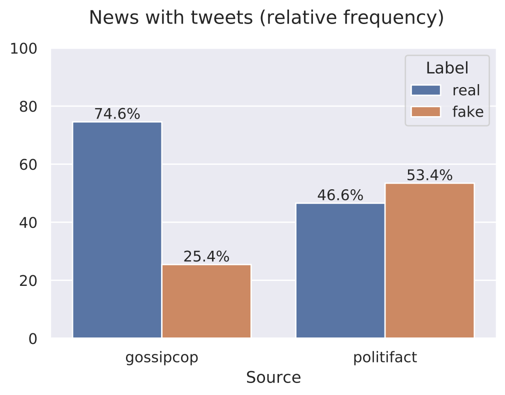


(b) Relative frequency.

Figure 3.2: News by label and source.



(a) Total count.



(b) Relative frequency.

Figure 3.3: News with at least one tweet by label and source.

If we take a look at the publish date, which is not available for all news, we see in figure 3.4 that GossipCop news were mostly published between 2016 and 2018, and the same distribution is followed by real and fake news (recall that there are more real news). However, PolitiFact real news were made available more or less uniformly between 2006 and mid 2017, while fake news were published between 2014 and 2019, with a distribution similar to GossipCop fake news. This means that if we included the news publish datetime, a model could correctly guess the news class by looking at the publish date, which does not make sense since fake news exist since long ago, so **we should exclude this feature when training models.**

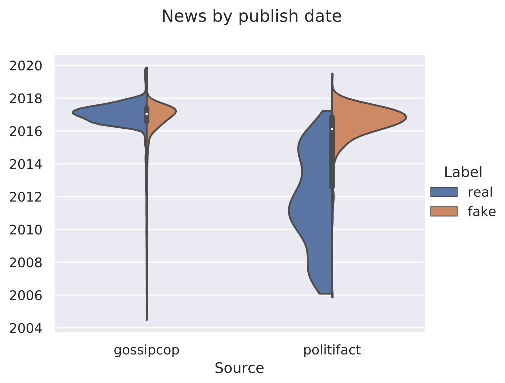


Figure 3.4: News by publish date.

Let us take the most frequent media sources (by looking at the URL domain) and see if there are differences regarding real and fake news publish datetimes in each media. We can see in picture 3.5 that most GossipCop sources have published mainly real news, but have a small fraction of fake news. However, *hollywoodlife* has published fake news, mostly. Regarding PolitiFact news, we notice that sources like *nytimes* or *whitehouse* have published almost no fake news, while news extracted from the *archive* were mostly fake, and it could be the reason why they were removed from the website. It is also curious that *washingtonpost* and *archive* have that high rate of fake news, especially compared to the GossipCop collection. To sum up, we see that **the URL domain might be a useful feature to detect which media are more prone to publishing fake news,** although it could be that the PolitiFact news collected by the dataset are not a representative sample from the real political news. We also notice here the importance of recovering news from the archive.

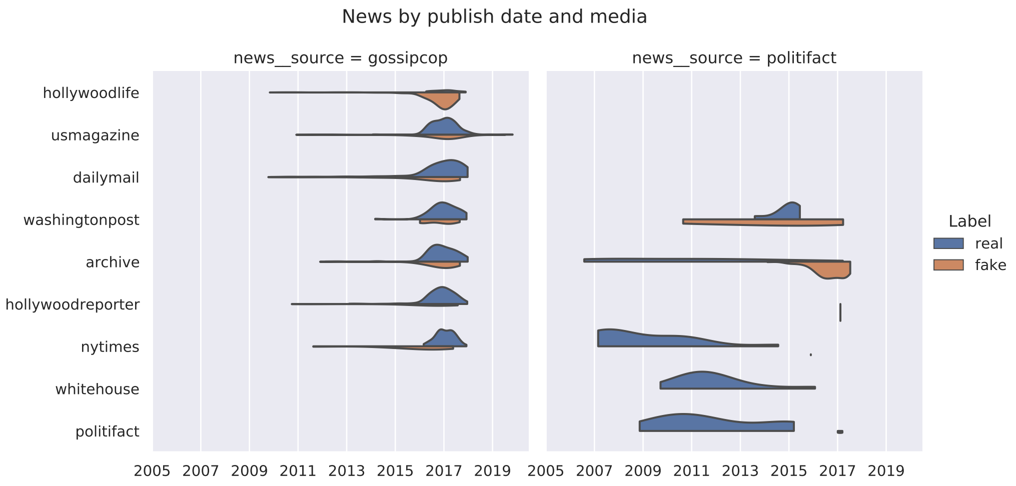


Figure 3.5: News by publish date and media. Areas represent relative frequency within each media.

Another detail that caught our attention was that, as seen in figure 3.6, **there are some GossipCop news that contain more than 400 pictures.** We know that FakeNewsNet authors used scraping tools to automatically extract features from news websites, but this is a strangely high number of pictures, even considering that the article might contain links to related news in a sidebar or at the end of the article. However, the distribution is similar for both real and fake news, so **this number could be less useful than knowing the publishing media, for instance.**

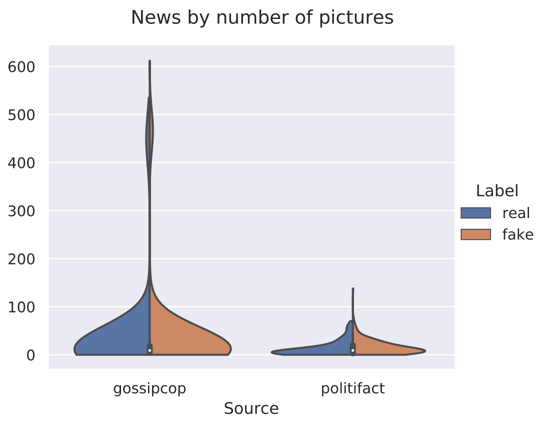
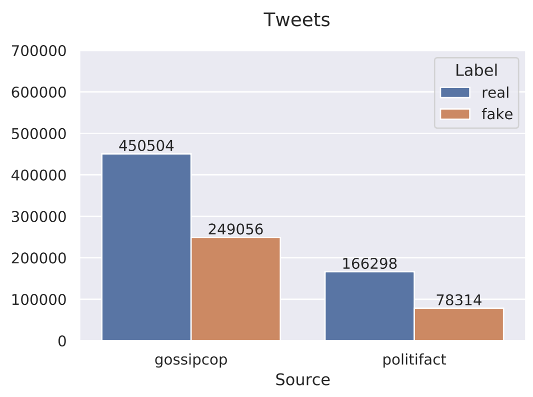


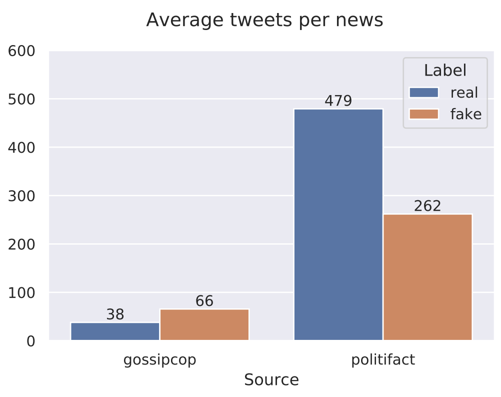
Figure 3.6: News by number of pictures. Each half-violin represents a normalized distribution function.

3.6.2 Tweet features

Let us put our attention now to the related tweets. As we can see in figure 3.7a, there are almost one million tweets related to training news, and PolitiFact news have many more tweets per news. It is also noticeable that in the GossipCop collection, fake news have more tweets in average than in PolitiFact news.



(a) Total count.



(b) Average tweets per news.

Figure 3.7: Tweets by label and source.

However, if we look further than just the average number of tweets, we can see in figure 3.8 many interesting facts that rebate the previous remarks. For instance, we notice that **the median number of tweets per news in GossipCop fake news is lower than in real news, and the upper tail is both wider and longer,** which ends up raising the average number of tweets. Curiously, **the opposite situation is observed in PolitiFact news,** with the upper tail being stronger for real news, although the median number of tweets is almost equal in both classes. We also notice that PolitiFact news have many more tweets compared to GossipCop news, and some PolitiFact news even have more than 10.000 news.

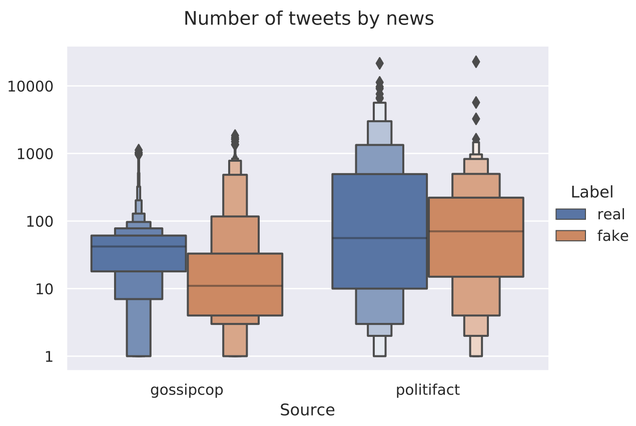


Figure 3.8: Number of tweets per news.

Looking at the publish date of tweets, we can see in figure 3.9 that **all the tweets were published before 2019.** However, we saw in figure 3.4 that some news were published in 2019, so this means that some news might be modified or corrected. Recall that in section 3.4, we removed tweets posted more than a day before than the associated news, but we did not remove any news, simply because we cannot be sure whether the presence of associated tweets is because that piece of news has been modified or the tweets were wrongly associated to the news. Therefore, this is a problem we could only avoid by manually reviewing all the news.

Regarding the tweet distribution, **we notice a high increase in tweets related to GossipCop fake news at the end of 2017,** whereas real news had a stable flow of tweets since 2016. Tweets related to PolitiFact also have a similar flow of tweets, with most tweets published in 2017.

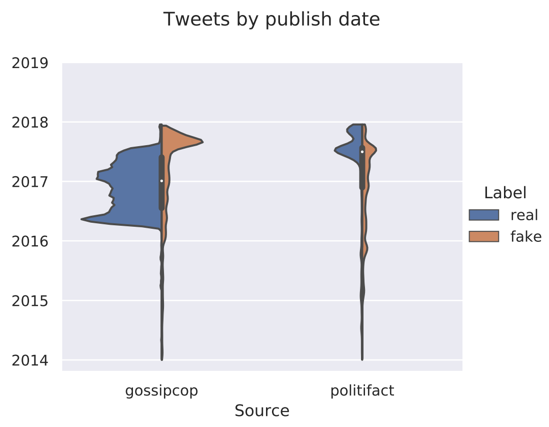


Figure 3.9: Tweets by publish date.

We were curious to see whether fake news had a smaller delay between the news publishing and the first tweet. Figure 3.10 proves us partly wrong, as it clearly shows that **50% gossicop real news had its first tweet published in the first 10 minutes, while the wait time was 4 hours in fake news.** PolitiFact news had much longer news-to-tweet times for real news, up to 1 month to cover the first 75%, although fake news follow a trend similar to GossipCop's. **Recall that some PolitiFact real were published long ago**, which explains these big differences.

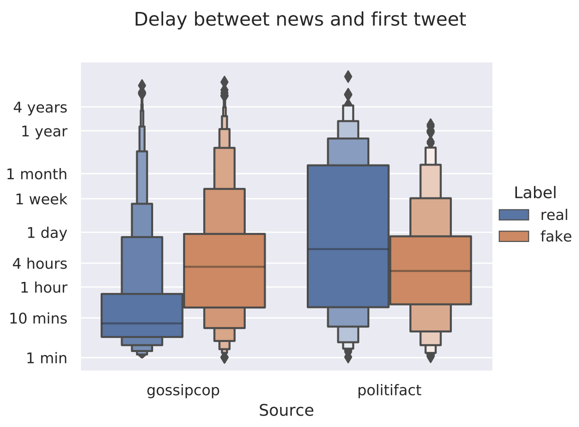


Figure 3.10: Delay betweet news and first tweet.

And if we take a closer look at which tools were used to publish tools, we discover in figure 3.11 that **GossipCop real news might be using automated tools based on *IFTTT* and *dlvr.it* to comment their news and gain visibility.** It could also be that some websites have an embedded Twitter section below the article where readers can directly write their reactions as a tweet.

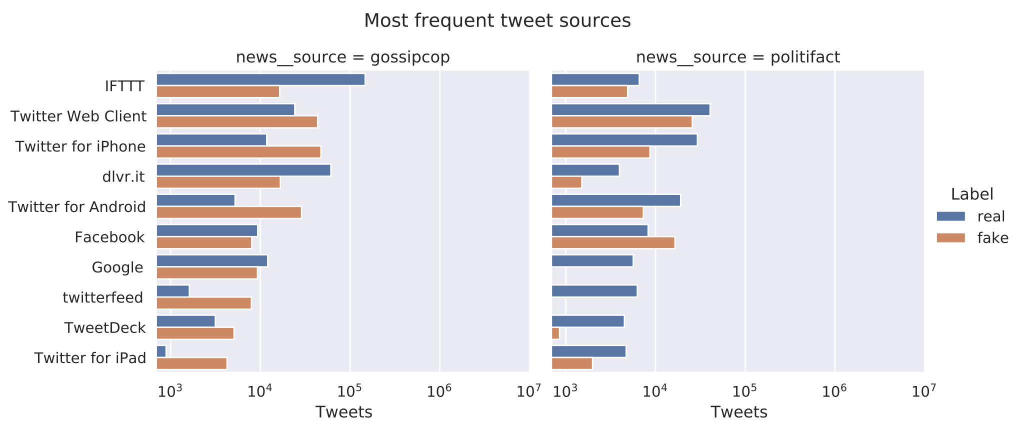


Figure 3.11: Most frequent tweet sources.

Regarding tweet languages, we can see in figure 3.12 that most of them are written in English, although there are some tweets written in other languages such as Japanese or Spanish.

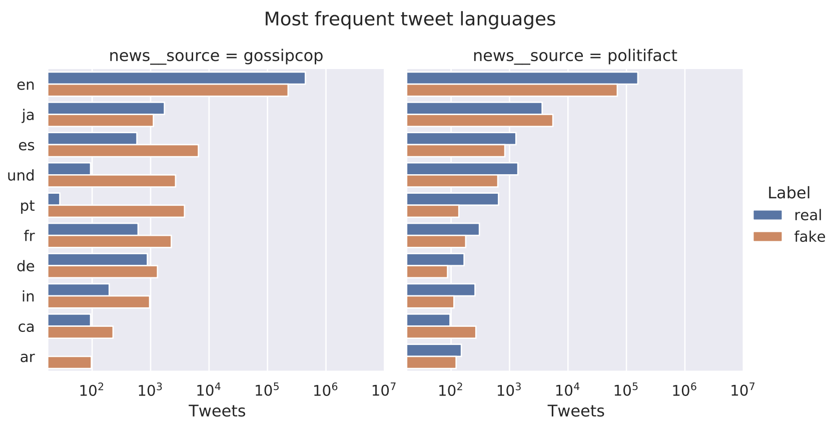


Figure 3.12: Most frequent tweet languages.

Similarly to languages, figure 3.13 shows that the most common tweet origins are English-speaking countries, especially the United States. However, **most tweets have this property empty, and it is interesting to note that, in the GossipCop collection, the country is specified in many more fake news tweets than those related to real news, even though there are many more tweets from real news.**

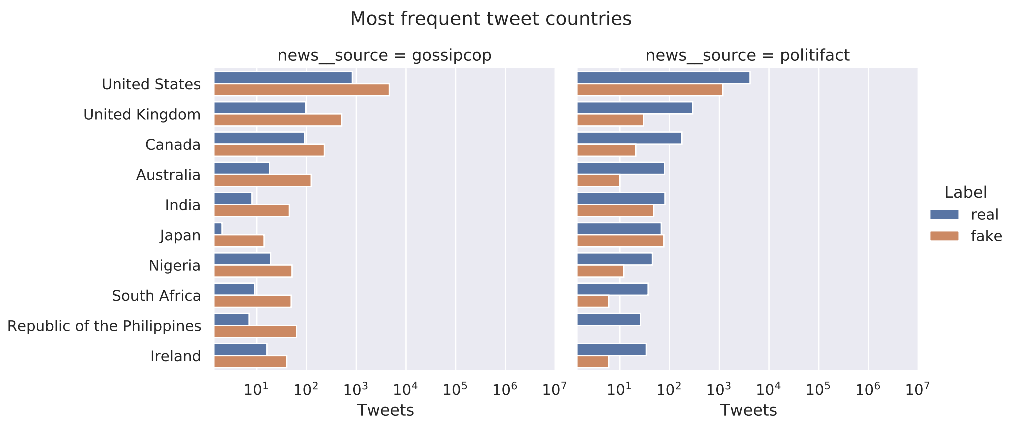


Figure 3.13: Most frequent tweet countries.

Next, we will see how are tweets distributed with respect to the number of likes. We can see in figure 3.14 that **most tweets have less than 10 likes,** although there are many tweets in both collections with more than 100 and 1.000 likes, which was to be expected. However, we did expect that most tweets would have more than 10 likes, and we can see that, in the GossipCop collection, tweets from real news have less likes than those from fake news, which might be a consequence of using automated tools to publish tweets.

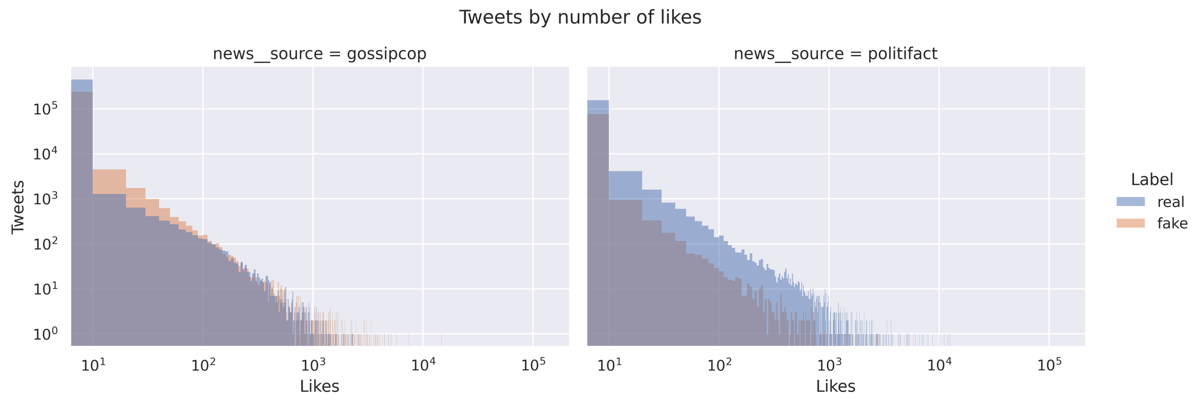


Figure 3.14: Tweets by number of likes.

Figure 3.15 shows a similar behaviour with respect to the number of retweets, although we can see that generally tweets have many more likes than retweets.

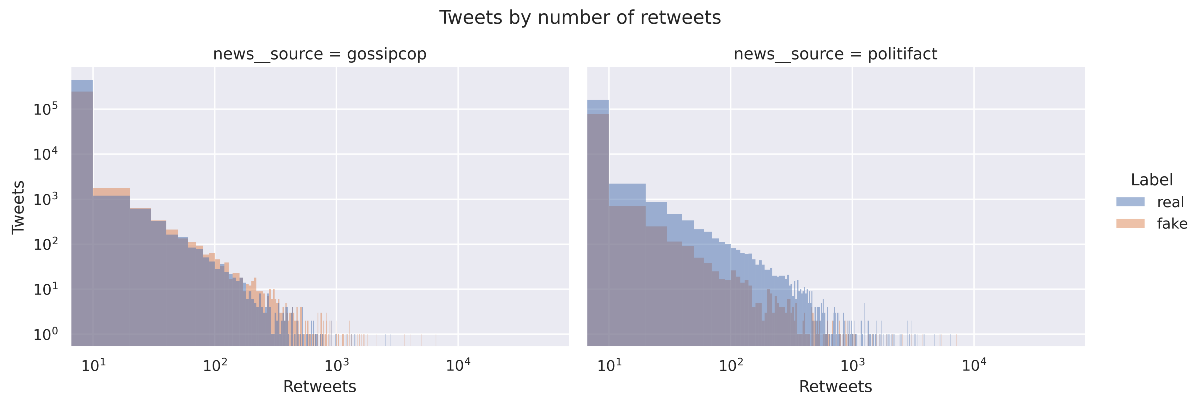


Figure 3.15: Tweets by number of retweets.

3.6.3 User profile features

Finally, we will take a look at features corresponding to the tweet authors. Figure 3.16 shows, for each label and source, the number of users that commented at least one news that belongs to that block. We see that, in the GossipCop collection, there are many more users involved in fake news than in real news, while the opposite is observed in PolitiFact.

We can see in figure 3.17 the distribution of user creation date. We can see that in the GossipCop collection, there are more users involved in fake news tweets than in real news, which suggest that **most real news tweets are published using a small set of accounts, and might be explained by the use of automated tools like IFTTT.** We also notice that there are some important spikes in 2013 and 2016 in the real news part, which might be connected to the previous observation, while all the other distributions are more or less uniform, with an initial spike in 2008, when Twitter became popular.

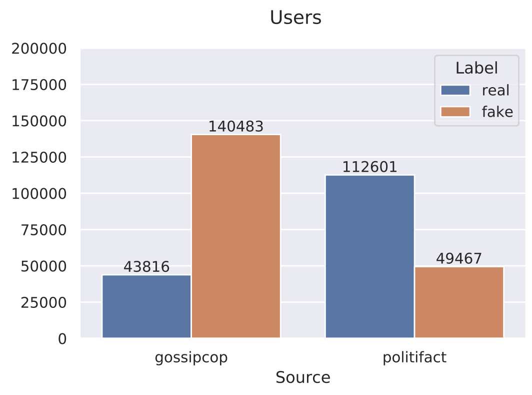


Figure 3.16: Users by label and source.

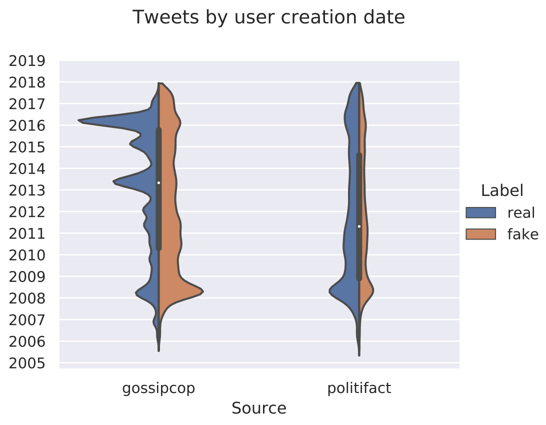


Figure 3.17: Tweets by user creation date.

It is interesting to plot the user creation dates depending on whether the profile has been modified or all the settings have been unaltered. Figure 3.18 shows that, in both collections, **most recent accounts have the default profile settings,** with an important spike in 2016, especially related to real news. On the other hand, **the number of created accounts with customized profiles has been going down since 2013,** which might be the actual trend for non-bot users, because Twitter's popularity has decreased in recent years. Therefore, we consider this feature might be quite useful to distinguish if a user corresponds to a real person or not, and therefore might help detecting fake news.

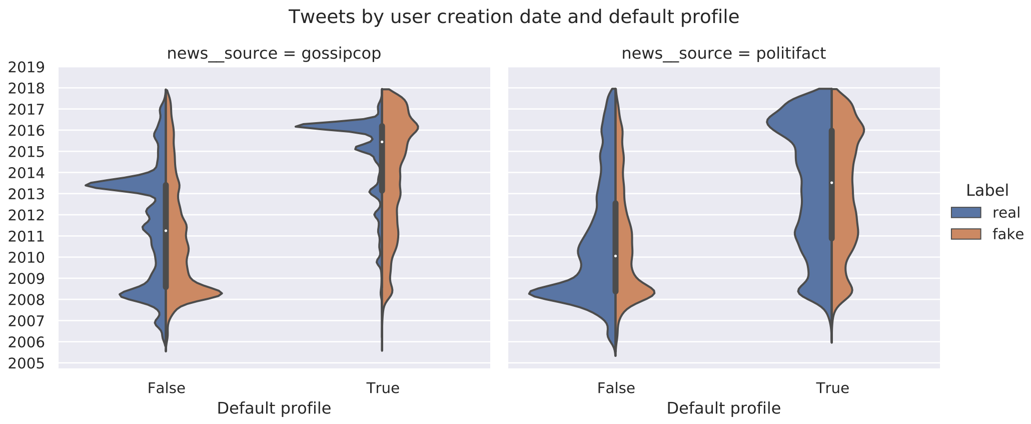


Figure 3.18: Tweets by user creation date and default profile.

Another measure we can analyze is the time from the user creation until each tweet was published. If bots are being used, we should expect shorter times. Figure 3.19 shows a natural increase each year, which is to be expected when users publish tweets regularly, but we notice that in GossipCop real news, **the median time to tweet is stabilized from 2016 to 2018, which indicates that many new accounts were created in that period, and this could be linked to the use of bots.** Furthermore, the opposite situation happens with fake news, with a considerable increase from 2016 to 2017, which might also indicate that bot accounts had been previously used and they were no longer needed. Regarding PolitiFact news, we notice that from 2016 to 2018, the increase has been slightly lower for fake news, which might also indicate the use of bots.

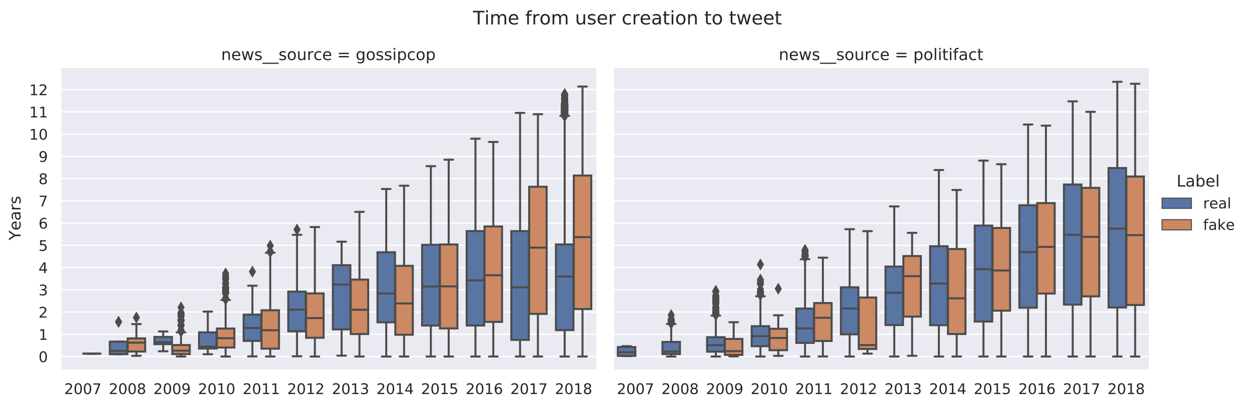


Figure 3.19: Time from user creation to tweet.

Let us now analyze some features of the user profiles. Figure 3.20 shows that many users do not have a specified location, but **those who had this field filled in are mostly from regions of the United States.** However, unlike what we saw in figure 3.13, there are more real news' tweets with this property filled than fake news' tweets.

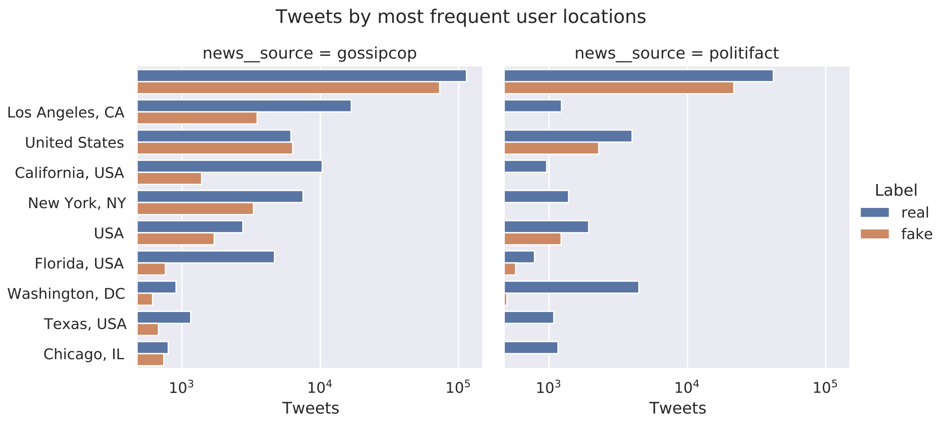


Figure 3.20: Tweets by user location.

Looking at figure 3.21, we see that **users that posted tweets about GossipCop real news in 2017 and 2018 had very few liked tweets,** which could be explained by the spike of users created in 2016, which also had the default user profile options. This phenomenon is also partially observed in GossipCop fake news, where the lower 50% also have few liked tweets. Users that posted in PolitiFact news appear to have more reasonable numbers.

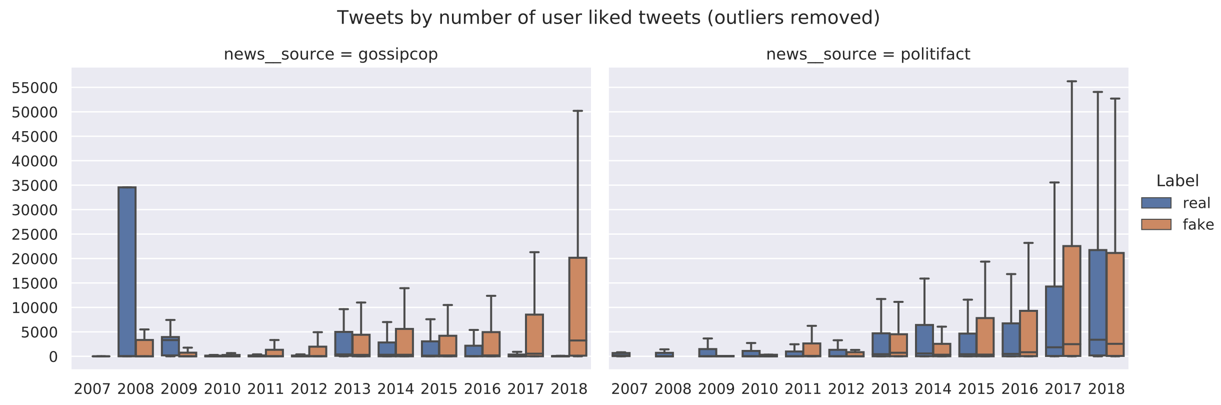


Figure 3.21: Tweets by number of user liked tweets. Outliers have been removed for clarity.

However, if we look at the number of user friends in figure 3.22, we do not see such differences as with the number of liked tweets. **Perhaps these bots or fake accounts try to simulate that they are real** by engaging other users and adding other accounts as friends, to have credible statistics, but did not even try to simulate a normal behaviour with respect to other tweets.

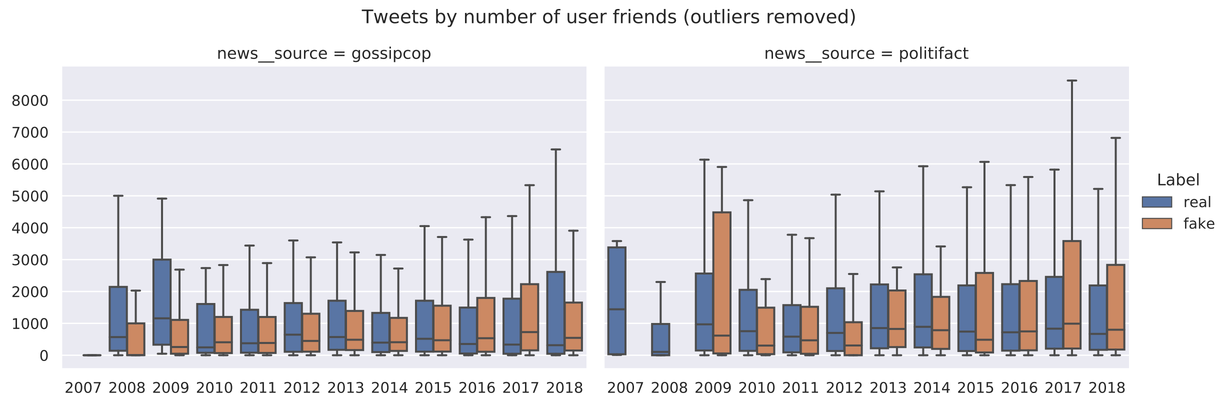


Figure 3.22: Tweets by number of user friends. Outliers have been removed for clarity.

3.7 Feature correlation

To finish this chapter, we compute feature correlations on the training set to gain more insights on how the features are correlated and, especially, if any of them has a greater correlation with class labels that might be especially useful when training models.

Each feature is considered as numerical, categorical or textual, following the feature types in tables 3.9, 3.10 and 3.11. We firstly compute the correlations of numerical and categorical features, and then analyze textual features separately. **News label will be considered here as a categorical feature.**

We will compute a correlation matrix for news features, another for tweet features and another for user profile features. Furthermore, we will compute the correlations separately for GossipCop and PolitiFact news. To compute tweet and user feature correlations, we will consider each tweet as an independent sample (as if each tweet was related to a different piece of news). We are aware that this assumption does not hold, since tweets are obviously not independent, but we have no other option if we want to use all the information available. Our goal is just to obtain an insight on possible interesting correlations.

For each pair of features, we will compute one of the following correlation measures, depending on the type of each feature.

- **Numerical-numerical.** We compute *Spearman's* correlation coefficient, which is simply Pearson's correlation applied to the ranks of the samples instead of their values. This correlation measure captures monotonic correlations, whereas Pearson's coefficient measures linear correlation. It is more informative than Pearson's correlation because we saw that many features have an exponential scaling.

- **Numerical-categorical.** We use the *correlation ratio*. Given a numerical feature X and a categorical feature Y, the correlation ratio of X and Y is defined as:

$$\eta(X,Y) = \sqrt{\frac{\sum_x n_x(\bar{y}_x - \bar{y})^2}{\sum_{x,i}(y_{x,i} - \bar{y})^2}},$$

  where

$$\bar{y}_x = \frac{\sum_i y_{x,i}}{n_x}, \qquad \bar{y} = \frac{\sum_x n_x \bar{y}_x}{\sum_x n_x},$$

  and with $y_{x,i}$ being the value of the i-th observation in the group $X = x$, and n_x the number of observations in that group. As we can see, the correlation ratio computes the weighted dispersion of the group averages divided by the total dispersion. When all the group averages coincide with the global average, the correlation ratio will be 0, meaning that knowing the measurements X gives no information about which groups Y they belong to. The maximum value is 1, when all the measurements are equal for each group, therefore knowing X completely determines Y.

- **Categorical-categorical.** We compute *Theil's U* statistic, also known as the *uncertainty coefficient*. This measure computes how much information is gained about one feature when the values of another feature are known. More precisely, given two features X,Y, we define:

$$U(Y|X) = \frac{H(Y) - H(Y|X)}{H(Y)},$$

  where

$$H(Y) = -\sum_y p_y \log p_y, \qquad H(Y|X) = -\sum_{x,y} \frac{p_{y,x}}{p_x} \log\left(\frac{p_{y,x}}{p_x}\right),$$

  are the *entropy* of Y and the *conditional entropy* of Y given X. Looking at the formula, we see that the statistic computes the entropy decrease in Y when X is known. For instance, if X defines a finer partition than Y, then $U(Y|X) = 1$, because X gives all the information about Y. And if X defines a random partition, then we should obtain $U(Y|X) \approx 0$, since no new information is gained.

Note that Spearman's correlation coefficient is symmetrical, this is, the value does not change if we swap the features. Correlation ratio needs to specify which feature is the categorical and which is the numerical, so we will assign the same value to each pair, independently of the order. However, Theil's U is not symmetrical. We will follow the convention that the cell (i, j) of the correlation matrix contains the U statistic of the i-th feature given the j-th feature, assuming that both are categorical. **In other words, we compute the statistic of the row features given the column features.**

3.7.1 Numerical and categorical features

Let us begin with GossipCop news features. We see in figure 3.23 that there are two big groups of correlated features, the top left involving the URL parts of media images and their main address (which usually coincide because many websites host their images in their own domain), and the bottom right with features about the embedded movies. The number of tweets and publish datetimes are left apart and only have small correlation with other features. Regarding news labels (first row), **we see that features are not correlated with the label, except the domain and subdomain of URL and images, and the number of tweets.** This does not mean that there is no connection between other features and news labels, but it might be more complex than a monotonic or linear correlation.

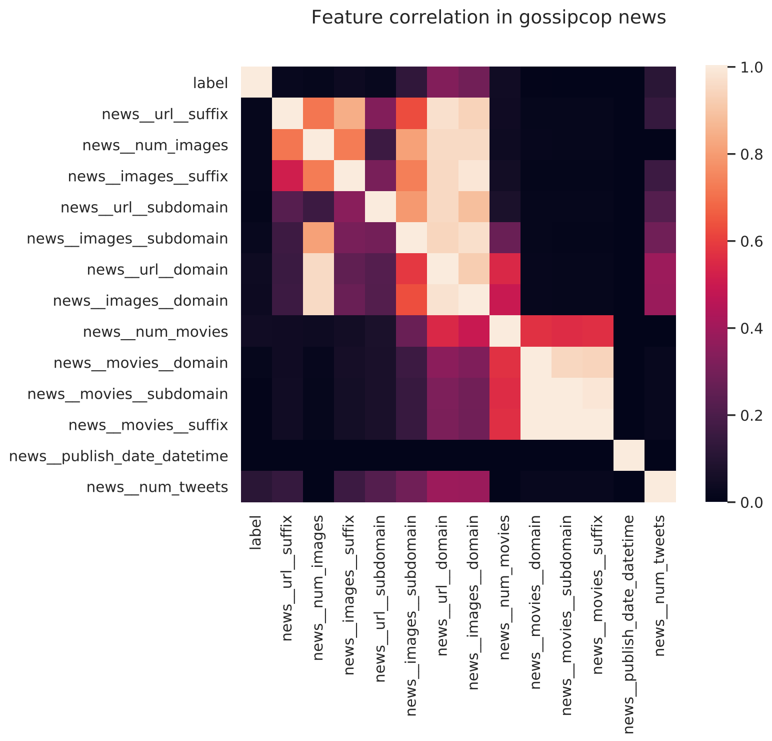


Figure 3.23: Feature correlation in GossipCop news.

Regarding PolitiFact news, figure 3.24 shows a similar situation, with the same two groups of correlated variables (with a different ordering, though), but **we notice that the features that were correlated with news labels in the GossipCop collection are now even more correlated.** This means that knowing the publishing media gives more knowledge about the label, or in other words, the proportions of real and fake news for each media are more extreme than in GossipCop news.

Moving to GossipCop tweet features, figure 3.25 shows that these features are much less correlated. There is a group of highly correlated features, involving the URLs contained in the tweet entities. Regarding news labels, we see that they are mostly correlated with the features in this group, with the number of user mentions, with the tweet source and with the hashtags contained in the tweet. **This means that the features given by Twitter to enrich the text and engage with other users, and the tools used to publish the tweet are also important.**

PolitiFact tweet features behave in a similar fashion, as we can see in figure 3.26, with the URL entities highly correlated between themselves and with the news labels. **However, user mentions and tweet sources are less correlated, perhaps indicating that less engagement is used in this collection.**

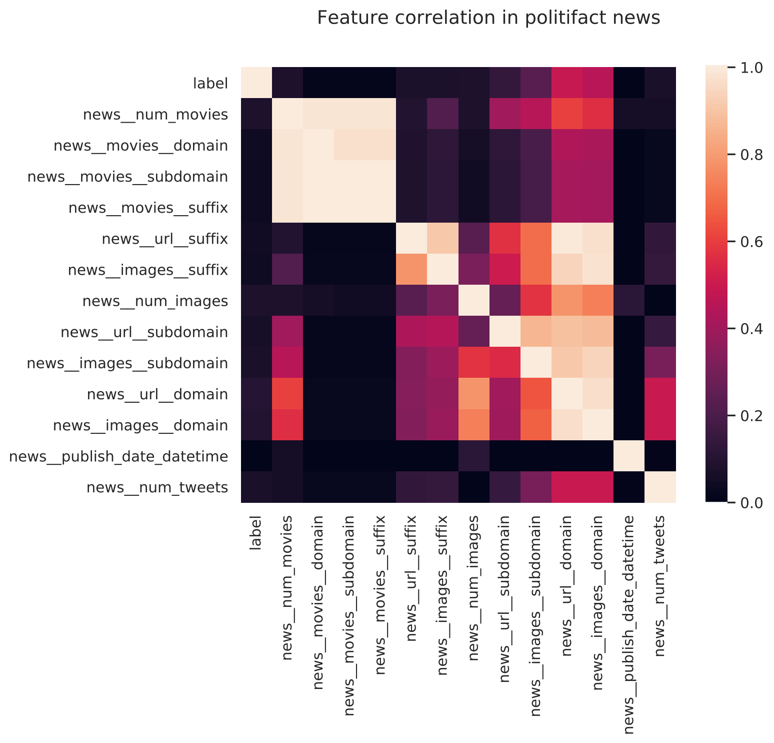


Figure 3.24: Feature correlation in PolitiFact news.

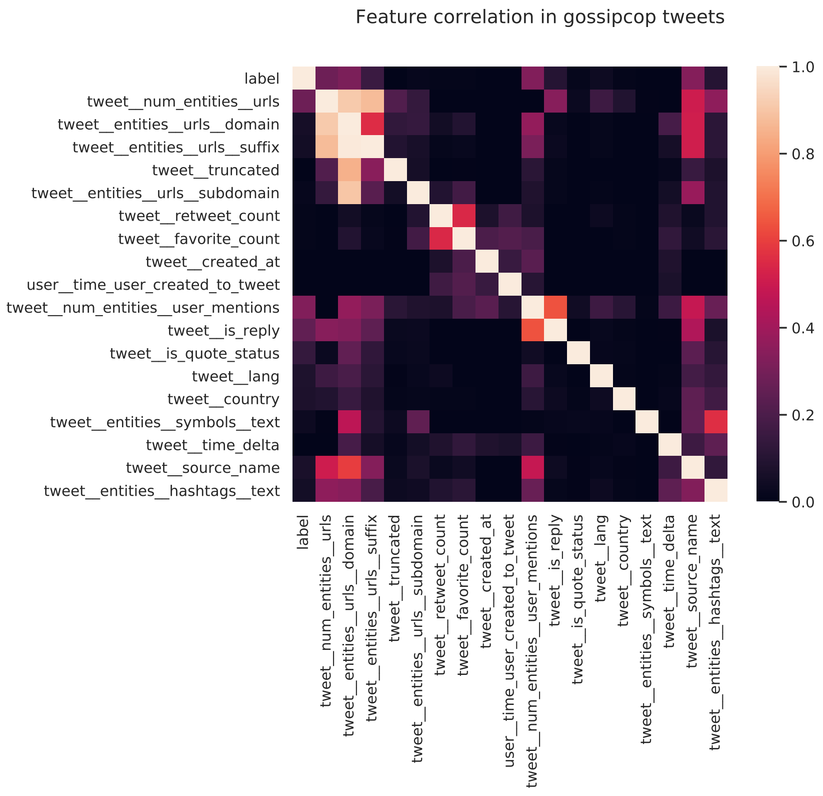


Figure 3.25: Feature correlation in GossipCop tweets.

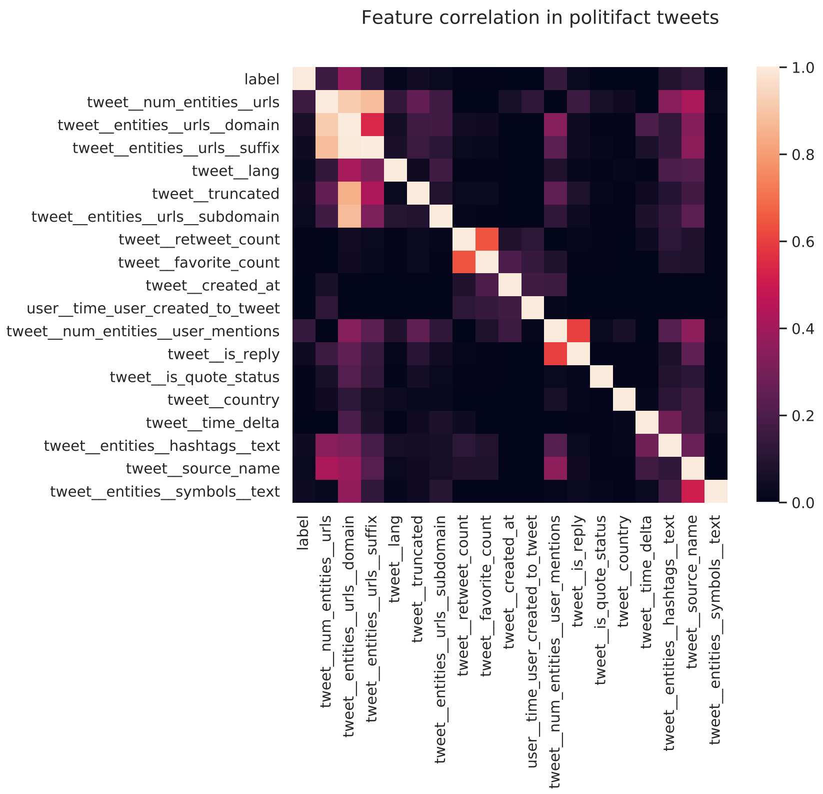


Figure 3.26: Feature correlation in PolitiFact tweets.

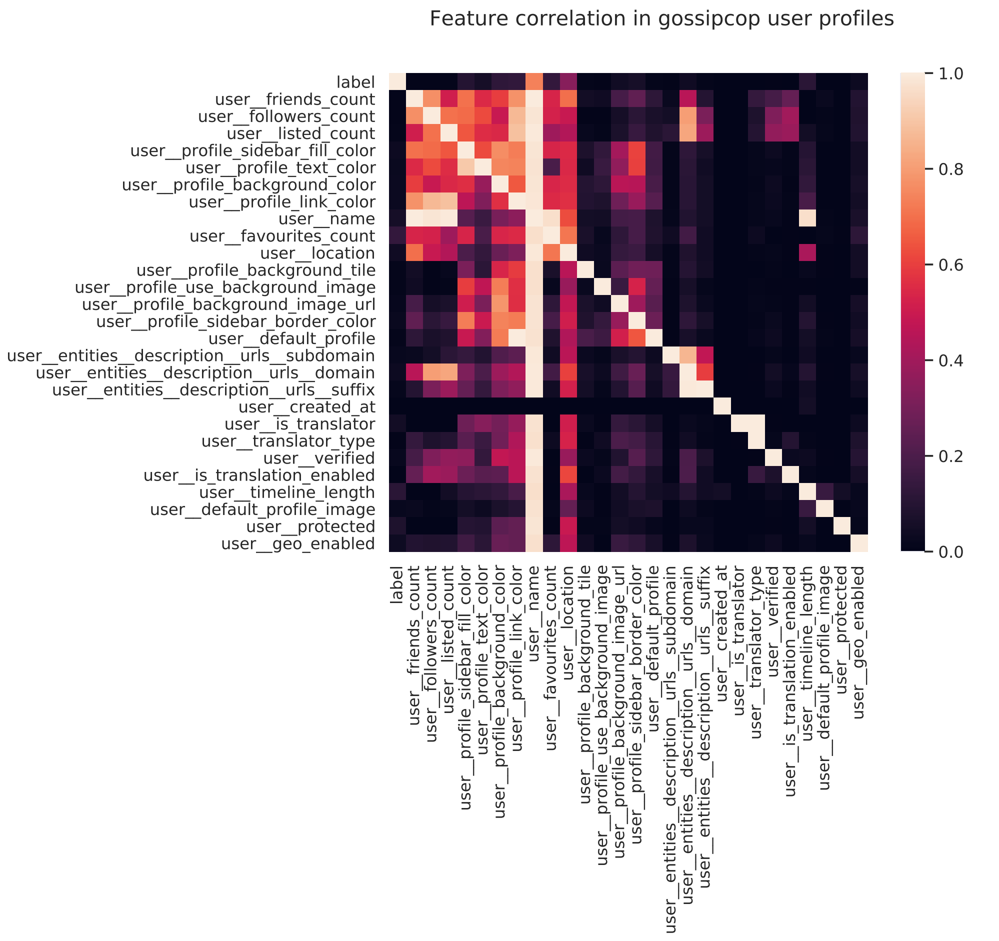


Figure 3.27: Feature correlation in GossipCop users.

We continue with GossipCop user profiles in figure 3.27. There is a large group of highly correlated features, some of them about user statistics and some other about profile customization. This might be related to figure 3.18, when we saw that many users have never modified their profiles and might be bot accounts. Regarding class labels, we see that user names are highly correlated with news labels, which indicates that **many users have a tendency to post mostly on fake news or mostly on real news.** However, we will not use user names to train models, because it does not make sense to classify news as fake or as real depending on a name. We will instead use profile features. Another feature that is correlated with news labels is the user location, but we have to keep in mind that this field is not exactly categorical, as each user is free to input any text string. Therefore, we should take this correlation with caution. As we will see in chapter 4, we will limit categorical features to the 100 most frequent values, which should solve this problem. The remaining features are correlated very little to labels.

Finally, figure 3.28 shows that PolitiFact user profile features have similar correlations, although the big group that we saw in figure 3.27 is now scattered, meaning that **profile customization might give less insights in this collection.** Features other than user names and locations are even less correlated with news labels.

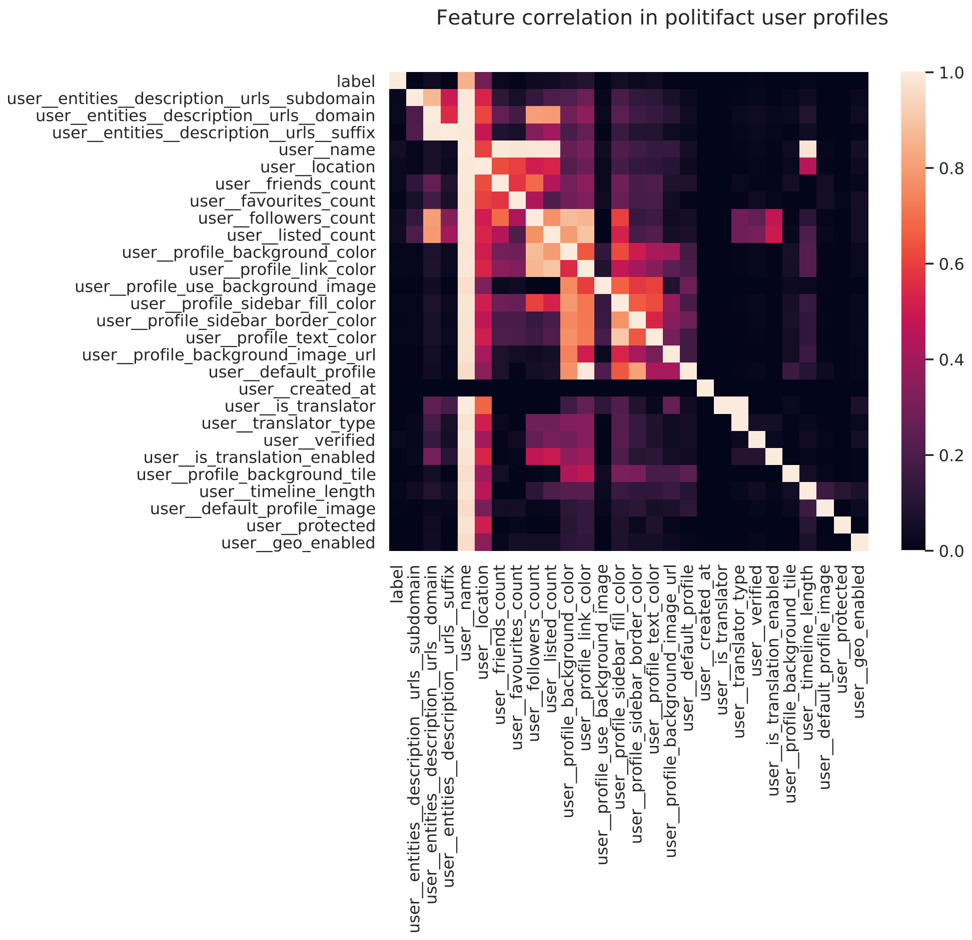


Figure 3.28: Feature correlation in PolitiFact users.

3.7.2 Textual features

To finish this section, we will analyze the correlation of textual features with news labels. Following the steps below, **we will obtain a correlation measure for each token present in each feature,** and we will plot the 50 most relevant tokens. We will apply the process for each collection separately.

1. For each textual feature, we will tokenize each text string and compute the TF-IDF representation of each sample. The tokenization will consist on lowercasing, removing punctuation and splitting by whitespaces, except for the tweet text, where we will use `nltk.tokenize.TweetTokenizer` and a regular expression to remove URLs.

2. Next, for each textual feature and token, we will compute the correlation ratio of the news labels (acting as categories) and the token's TF-IDF vector (numerical measurements). At the end, we will have vector for each feature, with each cell measuring the correlation of each token with the class labels. If a token does not appear in a feature, its value will be undefined.

3. We will keep only tokens appearing at least in the news text, tweet text or user description. We do this because these are the most important textual features.

4. Finally, we set undefined values to 0, divide each value by the maximum of its feature, compute the harmonic mean across features and take the 50 tokens with highest harmonic mean. We use the harmonic mean because we are more interested in tokens that are present in most features, rather than features that are very important only in one feature.

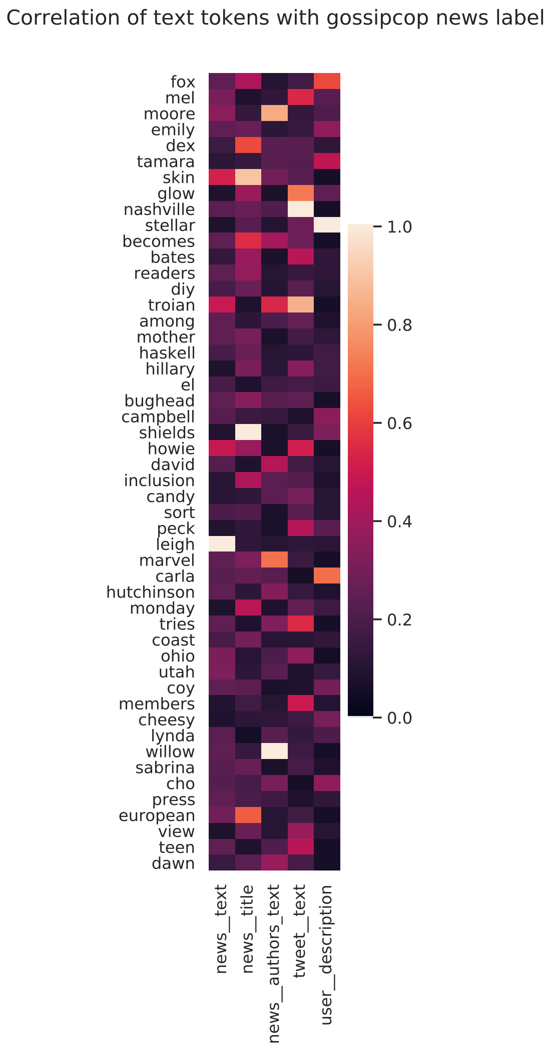
As we see in figure 3.29, **relevant tokens are quite different for each collection.** GossipCop's most important tokens include words related to:

- **Celebrities:** mel, tamara, hillary, campbell, sabrina.
- **Fashion:** skin, glow, stellar, diy.
- **United States' locations:** nashville, coast, ohio, utah.

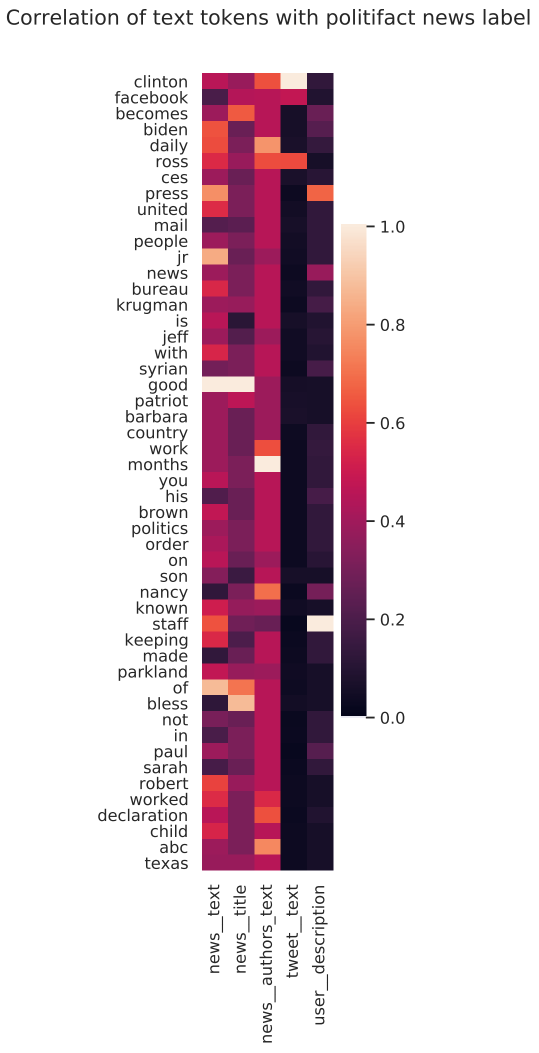
While PolitiFact tokens include words associated with:

- **Famous politicians:** clinton, biden, barbara, nancy.
- **Communication:** facebook, press, mail, news.
- **Political terms:** bureau, syrian, country, work, politics, staff, declaration.

Furthermore, the importance distribution across features is different in each collection. Figure 3.29a shows that, in the GossipCop collection, tokens are mostly important in one or two features each, but not in the others, and some tokens are relevant only in the tweet text or user description (for instance, *mel*, *glow*, *nashville*, *troian*, *fox* and *stellar*). However, most tokens in the PolitiFact collection (figure 3.29b) are important in all the news textual features but not in tweets or user descriptions (except *clinton*, *facebook*, *ross*, *press* and *staff*) **This indicates that tweets and user descriptions might be more useful in the GossipCop collection.**



(a) GossipCop.



(b) PolitiFact.

Figure 3.29: Correlation of textual features with news labels.

Chapter 4

System Architecture

In chapter 3, we downloaded the FakeNewsNet dataset and, after some cleaning and feature engineering, we ended with a train, validation and test set of news. Each piece of news is associated with a number of tweets, and we extracted the most relevant information from news, their related tweets and the users that posted each tweet. Next, we carried out an exploratory data analysis on the train set, where we noticed that tweets and user profiles might contain useful information to detect fake news. This exploratory analysis, along with the feature correlation matrices, highlighted some features with high class bias that might lead to spurious models, like the news publish date in the *PolitiFact* collection.

In this chapter, we define the architecture of the fake news detection systems that we test in chapter 5. We use a Deep Learning architecture that is capable to handle the one-to-many relationship between news and tweets. Moreover, we will see that the proposed architecture is very flexible and extensible, and can be a starting point to create more powerful models upon the state-of-the-art models mentioned in section 2.1. The two main goals of our architecture are:

- Handling varying-length sequences of tweets without the need of padding or truncating, and being able to apply reduction operations more complex than the sum or mean. In other words, our architecture accepts non-tabular input data, as long as the structure is known. In particular, it is possible to include trainable recurrent layers that extract information from ordered tweet sequences.

- Being able to choose the text representation technique and the subset of features that are used as input data, in a way that allows to fairly compare the results of different combinations. We use Vector Space Models (bag of words, frequency count and TF-IDF) and pretrained word embeddings: Word2Vec (see [Mik+13]) and SentenceBERT (see [Dev+18], [RG19]).

We will firstly train some well-known non Deep Learning algorithms, using only news features, to obtain an initial performance baseline. We will spend much less time on these models (we will not perform hyperparameter searches, for instance), since we only want to have some metrics of what can be achieved with standard non Deep Learning algorithms, beyond a *dummy* baseline.

This chapter is split in two sections: section 4.1 explains the non Deep Learning architecture, while section 4.2 describes the Deep Learning architecture. The actual implementation of each architecture (frameworks, packages) will be described with detail in appendix A.

4.1 Non Deep Learning

As explained earlier, we train non Deep Learning models to obtain a performance baseline of what can be achieved with standard classification algorithms. Here we only describe the architecture, the implementation details are left in appendix A.1.

Note. As we explain in the appendix, the architecture is implemented as a pipeline of two steps, the first being the preprocessing and textual vectorization procedure and the second being the classification algorithm. However, we will describe the preprocessing process assuming that it is a separate step that is performed before training the algorithms.

We want to test well-known non Deep Learning classification algorithms trained on news from each source separately, using different subsets of features and different textual representation techniques. **In other words, we want to try a list of *combinations* based on the following choices:**

- **Source:** GossipCop or PolitiFact.
- **Feature type:** only numerical, only categorical, only textual or all features.
- **Textual representation:** bag of words, frequency count or TF-IDF.
- **Classification algorithm:** logistic regression, linear SVM, radial basis function SVM, random forest, LightGBM or XGBoost.

The rest of the section explains each part of the architecture, in order. Firstly, we explain in 4.1.1 the input data used for fitting and evaluating models. Then, we describe the preprocessing and textual representation techniques in 4.1.2. We finish in 4.1.3 by listing the classification algorithms considered for testing.

4.1.1 Input data

We use as input data the train and test sets of news obtained in section 3.5, with the news features specified in table 3.9. **Tweets and user profiles are not used with non Deep Learning models.** Models are fit using the training set and evaluated on the test set. The field `label` contains the target labels, and represents news veracity, with real news encoded as 0 and fake news as 1.

Note. The validation set is not used, since we do not perform hyperparameter tuning. However, we do not introduce these news in the train or test set because we want to train and test all the models using the same sets, and we use the validation set in the deep learning architecture.

During each train-test process, **we train using training news from GossipCop or from PolitiFact, and evaluate using test news from the same source.** We do not use both sources at the same time, because there are many more news from GossipCop, so the results would be similar to using GossipCop only.

Furthermore, we want to **test models trained on subsets of features depending on their type:** only numerical features, only categorical features, only textual features or all features.

Excluded features

The following features from table 3.9 are excluded from the fitting and evaluating process:

- `news__source`. Contains the news source (GossipCop or PolitiFact). Since we train separately on both sources, this feature is constant on each source and can be removed.
- `news__publish_date_datetime`. We exclude publish datetimes because they are not available for all news and because class bias is very high with respect to the publish date, especially in the PolitiFact collection. More precisely, we assume that there will always coexist real and fake news in a similar proportion and, therefore, models should not use absolute datetimes to take advantage of the dataset's class bias with respect to publish dates. We do not deny that there might exist a time-dependent class bias, but we assume that the bias found in the exploratory data analysis is only observed in this dataset.

4.1.2 Preprocessing and textual representation

Before fitting and evaluating, both the train and test set need to undergo a preprocessing process where:

- Numerical features are centered and scaled. This is required when using algorithms like logistic regression and support vector machines, that can be affected by feature scale, and does not harm with random forest or tree-based gradient boosting methods.

- Categorical features are encoded as one-hot vectors (also called *dummy* features). This is the usual way of encoding categorical features.

- Textual features are represented as vectors. In this non Deep Learning architecture, we only use Vector Space Models, where we first tokenize the original strings and then represent each string as a vector. We tokenize strings by lowercasing, removing punctuation and splitting by whitespaces, and then compute one of the following representations (we test the effect of using each one of these representations in chapter 5):

  - **Bag of words.** Also called *binary* vectorization. Indicates the presence or absence of each token in the vocabulary, in the given string.
  - **Frequency count.** Indicates the frequency count of each token.
  - **TF-IDF.** Indicates the frequency count of each token, multiplied by a correction factor inversely proportional to the number of strings where that token appears.

  We create the vocabulary using the 5.000 most frequent tokens, which seemed a reasonable size during our initial tests. Each textual feature has its own vocabulary.

Note. The preprocessing steps use only the train set to compute mean and variance of numerical features, find the categorical features' values and create textual features' vocabulary. The test set is only transformed using the information contained in the train set, so that no information is leaked to the model.

4.1.3 Classification algorithms

We want to obtain a baseline using different classification algorithms. We consider this selection to be sufficiently varied, since it contains some of the most used and successful algorithms. The selected algorithms are:

- **Logistic regression.**

- **Support vector machines.** Using the linear kernel and the radial basis function kernel.

- **Random forest.**

- **Gradient boosting methods.** LightGBM and XGBoost algorithms are used.

4.2 Deep Learning

The Deep Learning architecture is a deep neural network that contains a mechanism to handle the one-to-many news-tweets relationship. The actual implementation is explained in appendix A.2.

Note. As we explain in the appendix, the preprocessing and textual vectorization are implemented as the initial part of the deep neural network, although in this section, we will just assume that preprocessing is performed before training the deep network.

The core idea is that, when the (preprocessed) data enters the network, news features are concatenated with a vector, computed by the deep network, that is a summarization of related tweets and the user profiles of the authors of those tweets. This summarization or *reduction* can be a trainable part of the network, unlike usual reductions like the mean or maximum, and can handle varying-length sequences of tweets without padding or truncating to a fixed length. This allows representing all the information associated to a piece of news as a one-dimensional vector. These vectors are then passed to a prediction head composed of several dense layers with regularization, and the network ends with a one-cell dense layer with sigmoid activation function, which outputs the probability of the news being fake.

We want to test our Deep Learning architecture training on news from each source separately, using different subsets of features based on their type and origin, and different textual representation techniques. **We will try a list of *combinations* based on the following choices:**

- **Source:** GossipCop or PolitiFact.
- **Feature type:** only numerical, only categorical, only textual or all features.
- **Feature origin:** news, tweets, user profiles, news and tweets, news and user profiles or all of them.
- **Textual representation:** bag of words, frequency count, TF-IDF, Word2Vec or SentenceBERT.

In the rest of the section, we explain each part of the architecture, following the data flow order. We start by explaining the input data in 4.2.1. Next, we describe the preprocessing process and textual representation techniques in 4.2.2. We continue with the summarization of tweets and user profiles in 4.2.3, and the prediction head in 4.2.4. Finally, we describe the training and evaluation process in 4.2.5

4.2.1 Input data

We use as input data the train, validation and test sets obtained in section 3.5, with the features specified in tables 3.9 (news), 3.10 (tweets) and 3.11 (user profiles). Target labels are contained in `label`, with real news encoded as 0 and fake news as 1. As in the non Deep Learning architecture, we will only use news from one source at a time, and the 80-10-10 proportions are preserved because the splits are stratified by label and source.

Note. The validation set is only used to tune the classification threshold. We do not perform any other hyperparameter tuning, although we did some initial tests to find a set of hyperparameters adequate to carry out all the covered combinations in similar conditions.

Due to the high amount of Twitter data, **we will only use up to the first 100 tweets (in chronological order) for each piece of news.** We saw in the exploratory data analysis in 3.6.2 that some news have more than 10.000 tweets, which needs a lot of computational power when using SentenceBERT. However, 92% news have at most 100 related tweets, and we consider that it is a reasonable trade-off between performance and resources.

We are interested **in testing models trained on subsets of features** depending both on their type (numerical, categorical, textual) and on their origin (news features, tweet features, user profile features).

Note. Whenever we include tweet or user profile features, we will not train with news that have no associated tweets. However, as mentioned in chapter 5, when using only news features, we will try both using only news with tweets and using all news, to check if there is any difference in results.

Excluded features

The following features from tables 3.9, 3.10 and 3.11 are excluded from the whole process:

- `news__source`. Same reason as in 4.2.1.
- `news__publish_date_datetime`. Exactly as in 4.2.1.
- `tweet__created_at`. Tweet posting dates. We replace this feature by `tweet__time_delta`, which contains the time difference between consecutive tweets of the same news, or between news and first tweet, if it is the first tweet. The reason is that we do not want the model to decide news veracity based on absolute datetimes. We do not deny that there might exist a time-dependent class bias, but we assume that the bias found in the exploratory data analysis is only observed in this dataset.
- `user__id`. Contains the user identifiers. It is needed to join the tweet with its author, but we must exclude it, since we want the model to detect fake news depending on the user profile features, rather than the identifier.
- `tweet__news`. Contains the news identifiers. Used to join news and tweets, and it is excluded for the same reason as `user__id`.
- `user__name`. User name. Does not have to be unique, but certainly has almost unique values, and we do not want the model to use this feature.
- `user__screen_name`. User screen name. Must be unique, and is excluded for the same reasons as `user__name`.
- `user__created_at`. User creation datetimes. We drop this feature and, instead, include the feature `user__time_user_created_to_tweet`, which calculates the time difference between user creation and tweet publish datetimes. The reasons explained with `tweet__created_at` apply here, too.

4.2.2 Preprocessing and textual representation

Before training deep neural networks, the train, validation and test sets need to go through a preprocessing process where:

- Numerical features are centered and scaled. This is not required by neural networks, but might improve convergence speed and numerical stability.
- Categorical features are encoded as one-hot vectors. **We will limit each one-hot encoding to only distinguish between the 100 most frequent values in the train set for that feature.** The problem lies mostly in user profile features, where some categorical features have more than 10.000 different values and may reduce the usefulness of other features.
- Textual features are encoded as vectors. We use both Vector Space Models and pretrained word embeddings. We can choose the representation technique, but the same technique will be used for all the textual features. Vector Space Models and Word2Vec embeddings will be preceded by a tokenization process similar to the non Deep Learning architecture: lowercasing, removing punctuation and splitting by whitespaces. They will also need to create a vocabulary for each feature, which will be limited to the 5.000 most frequent tokens. On the other hand, SentenceBERT contains its own tokenization and does not need to manually create a vocabulary. We will test the following textual representations in chapter 5:
  - **Bag of words.** Vector Space Model indicating the presence or absence of each token in the vocabulary.

- **Frequency count.** Vector Space Model containing the frequency count of each token.
- **TF-IDF.** Vector Space Model that represents each string as the frequency count of each token, multiplied by a correction factor inversely proportional to the number of strings where that token appears.
- **Word2Vec.** We use Google's Word2Vec pretrained word embeddings. Specifically, we use `word2vec-google-news-300`, which contain vector representations of length 300 that have been trained on text from Google News. Since there might be tokens that are in the features strings but not in Word2Vec's vocabulary, we need to first obtain the vocabulary of each feature in the training set and then select the 5000 most frequent tokens that are also in Word2Vec's vocabulary. Furthermore, we need to generate two random vectors for out-of-vocabulary tokens and empty strings, respectively. Note that Word2Vec assigns a vector to each token in the string. **To obtain a sentence representation, we will compute the mean of token representations.**
- **SentenceBERT.** There are several variations of SentenceBERT. We use the one based on BERT base model, trained on Natural Language Inference data, and taking as sentence representation the mean of token embeddings. SentenceBERT is a deep neural network itself, and has its own tokenization process. We use SentenceBERT to obtain a sentence representation of each string. Each string will be summarized as a vector of length 768.

Note. As in the non Deep Learning architecture, these preprocessing steps use only the train set to compute mean and variances, find the categorical values and create vocabularies. The validation and test sets are only transformed using the information contained in the train set.

4.2.3 Handling one-to-many news-tweets relationship

We handle the one-to-many relationship between news and tweets by summarizing the sequence of user engagements into a fixed-size vector, in a similar way as in [RSL17] and [SML19]. However, we take advantage of the advances in Tensorflow that allow to process non-tabular data, so that no reshaping has to be done before passing the data to the system.

For instance, the authors of [RSL17] and [SML19] propose to aggregate user engagements in temporal windows, and also need to compute the singular value decomposition of coincidence matrices which indicate what users comment on what news, or how many times two users posted a comment on the same news piece. Therefore, these systems are using much less information than our architecture, and they are losing information due to the aggregation in temporal windows.

In contrast, our architecture takes as input the whole, unaggregated sequence of user engagements, and applies a summarization layer that can be as complex as desired. In the implementation we test in chapter 5, which is described in detail in appendix A.2.2, we firstly concatenate the vector containing tweet features with the user profile features of the user who published the tweet. Then, **we compute the feature-wise mean of the concatenated vectors,** and we append the resulting vector to the news features. This final vector contains all the information from that news piece and its user engagements.

However, **we can also apply more complex summarizations, like a recurrent layer** that takes the ordered sequence of user engagements and outputs a vector representation of the whole sequence. In fact, we initially used a LSTM layer instead of the feature-wise mean, which could be potentially powerful to understand the engagement flow.

Using a summarization in this way, we can handle varying-length sequences of tweets without padding or truncating any sequence. What is more, our approach can be generalized to include any information stored hierarchically, as long as we know the structure. For instance, if we had information about the followers of each user, we would have a list of user profiles for each tweet author (therefore a varying-length list of varying-length lists of user profiles, for each piece of news), and could apply a summarization that includes information from followers. However, the computational power required grows exponentially, so it would be necessary to optimize the architecture by limiting the model complexity or using less features.

Our approach is also very flexible, in the sense that there are multiple ways of processing user engagements to obtain a summarized vector representation. For instance, we could have considered that each news has a *sequence* of tweets and a *set* of users, summarize both parts separately and then join the summarized vectors. In this case, the summarized vector from user profiles is obtained independently from the tweet features and with no temporal information with respect to the interaction order, therefore obtaining a representation of the involved users independently of their tweets. This summarization might be able to extract even more information, since user features are not masked by tweet features and vice versa, and could be useful to cluster the set of users depending on the type of news they engage with, similar to the coincidence matrix used by [RSL17] and [SML19] but using much more information about users.

4.2.4 Prediction head

The prediction head consists on a number of interleaving dense and dropout layers, followed by a final dense layer with one cell and sigmoid activation. The output is interpreted as the probability of the news being fake.

After some initial tests, we decided to settle with a prediction head with two dense-dropout blocks, followed by the final one-cell dense layer. The dense layers have 48 cells each, with *l2* regularization and *SELU* activation function, while the dropout layers have a dropout rate of 0.2. We use this prediction head in all tests.

Note. We achieved similar results with 64-cell dense layers, and with dropout rates slightly lower or higher. The regularization and activation are not decisive either, with similar results using no regularization or the *RELU* activation. Adding more dense-dropout blocks did not improve significatively, although the model did perform a bit worse with only one block, and when dropout layers were not used.

4.2.5 Training and evaluation

The training and testing procedure will be the following:

1. Select the input data (source and features) and the textual representation technique.
2. Fit the deep neural network for a number of *epochs* using the training set. In our initial tests, 10 epochs were enough for all combinations to converge.
3. Compute the predictions for the validation set, calculate the precision-recall curve (from now on, PR curve) and choose as optimal classification threshold the lowest value that maximizes F1 score.
4. Evaluate the model in the test set using this optimal threshold.

Note. As explained in the appendix A.2, we would have liked to use early stopping techniques to stop training when model's performance in the validation set no longer improves. However, this was not possible due to technical reasons.

We train the deep neural networks using the binary cross-entropy loss, since this is a two-class classification problem with 0-1 labels. We use a batch size of 32, except when using SentenceBERT, where we use 16 samples per batch. We use the Adam optimizer with a learning rate of 0.001, which seems a reasonable choice and reaches convergence in all the combinations with consistent results. We tried SGD and Adadelta optimizers, and the results were similar.

Chapter 5

Experiments and Results

We explained in chapter 4 our Deep Learning architecture, which can mix news features with tweet and user profile information and is capable of extracting information from the whole sequence of user engagements by fully handling the natural one-to-many relationship between news and tweets. Although our focus is on this architecture, we also briefly described a non Deep Learning architecture that we use to obtain a performance baseline beyond a *dummy* baseline. Moreover, it will serve to highlight some trends that will be ultimately confirmed by our Deep Learning models.

In this chapter, we describe, in section 5.1, the tests that we have carried out using news from each collection in the FakeNewsNet dataset and trying different subsets of features and different vectorization techniques. Then, we analyze, in section 5.2 the obtained results and discuss the reasons of those results and whether the particularities observed in the exploratory analysis are connected to the results.

The main goal of the experiments is to check whether using information from tweets and user profiles helps improve the performance of the tested models. Implicitly, this is related to the question of whether the proposed architecture is adequate for this task. We are also interested to see how each textual representation technique behaves, and finally, whether categorical and numerical features are useful along with textual features.

We only include in this chapter the main results, mostly the F1 scores obtained in the test set. Other measures such as accuracy, precision-recall AUC and ROC AUC can be seen in appendix B.

5.1 Experiments

We carried out a series of experiments, changing the input data and the textual representation technique. We firstly tested the non Deep Learning architecture, to obtain a performance baseline, and then the Deep Learning architecture. Finally, we performed an ablation test for the Deep Learning architecture, to understand which features are contributing more to the model.

The following subsections explain the combinations tested for each architecture. Each architecture has a different set of choices to be made (news source, features used, textual representation), and each choice has a set of possible values. **We tested each possible combination 5 times, using the same train, validation and test sets all the time, to obtain comparable results.**

Note. Recall that the Deep Learning architecture is designed to discard news without associated tweets, although it can use all news if only news features are included. We already saw in figure 3.3 that the class distribution in the subset of news with tweets was slightly different with respect to the whole set. We need to check if there are significative differences between using all news or news with tweets. We will do so in the non Deep Learning experiments.

For each training-testing process, we train the model in the training set, and evaluate in the test set, according to what we explained in chapter 4. With the Deep Learning architecture, we use the validation set to optimize the classification threshold, as explained in 4.2.5.

Regarding performance measures, the task of detecting fake news is a binary classification problem. The goal is to detect as many fake news as possible, without marking real news as fake. In other words, we want to achieve a high *sensibility* (or *recall*) with high *specificity*, often focusing in the former, since it is usually considered more important to block fake news. However, both measures are insensitive to class bias, which is especially present in the GossipCop collection, with 75% real news. Therefore, it is more informative to compute the *precision* (or *positive predictive value*), which is sensitive to class bias. For this reason, **we decided to focus on the F1 score,** which is the harmonic mean of precision and recall, and will be our main performance measure. **We also compute the accuracy and, in the Deep Learning architecture, the precision-recall AUC and ROC AUC,** to obtain an overall measure considering all possible classification thresholds.

5.1.1 Non Deep Learning

The tested combinations are selected by choosing a value for each of these choices:

1. **Source:**
    (a) GossipCop.
    (b) PolitiFact.
2. **Feature type:**
    (a) Only numerical features.
    (b) Only categorical features.
    (c) Only textual features.
    (d) All features.
3. **Textual representation:**
    (a) Bag of words.
    (b) Frequency count.
    (c) TF-IDF.
4. **Classification algorithm:**
    (a) Logistic regression.
    (b) Linear SVM.
    (c) Radial-basis-function SVM.
    (d) Random forest.
    (e) LightGBM.
    (f) XGBoost.

Note. When textual features are not used, the textual representation choice is ignored.

For instance, one combination could be to use GossipCop news, using only textual features, with TF-IDF representation and LightGBM as the classification algorithm. Another combination could be to select

PolitiFact news, using only categorical features (hence, we ignore the textual representation choice) and fitting a logistic regression.

We carried out the experiments twice: firstly, using all news, and then, using only news which have associated tweets. The results using only news with tweets will tell if there are significative differences when the rest of news are dropped, as it is the case with the Deep Learning architecture.

To sum up, there are 96 different combinations, each one is tested five times, and they are carried out twice. Therefore, we did a total of $2 \cdot 96 \cdot 5 = 960$ different experiments.

5.1.2 Deep Learning

Combinations are selected by choosing a value for each of these choices:

1. **Source:**
   (a) GossipCop.
   (b) PolitiFact.
2. **Feature type:**
   (a) Only numerical features.
   (b) Only categorical features.
   (c) Only textual features.
   (d) All features.
3. **Feature origin:**
   (a) News.
   (b) Tweets.
   (c) User profiles.
   (d) News and tweets.
   (e) News and user profiles.
   (f) News, tweets and user profiles.
4. **Textual representation:**
   (a) Bag of words.
   (b) Frequency count.
   (c) TF-IDF.
   (d) Word2Vec.
   (e) SentenceBERT.

Note. Again, when textual features are not used, the textual representation choice is ignored. Feature origin refers to whether we use features from the news, from the tweets or from the user profiles.

We carried out extra experiments using news features from all news, to check if there are significative differences with respect to using news with tweets.

There are 144 different combinations and each one is tested five times, so we did a total of $168 \cdot 5 = 840$ different experiments.

Finally, we performed an ablation test for each FakeNewsNet collection, using all features from all origins. Firstly, for each available feature, we fit a model using only that feature on the train set, find the optimal classification threshold on the validation set and evaluate the model on the test set. Then, we sort the features by their test F1 score in ascending order and fit models incrementally adding the features one by one, to see whether they improve model performance.

5.2 Main results

In this section, we present the main results obtained from the experiments described above. We start in 5.2.1 by calculating a dummy performance baseline. Then, we show in 5.2.2 the performance baselines obtained with the non Deep Learning architecture. We continue in 5.2.3 discussing the results obtained with the Deep Learning architecture. Finally, we analyze in 5.2.4 the ablation test performed with the Deep Learning architecture.

By main results, we mean the F1 scores obtained from the evaluation in the test set of each combination. As we explained earlier, our main performance measure is the F1 score, which we consider adequate to the problem. We leave in appendix B the test accuracy scores with both architectures, and the precision-recall and ROC AUCs obtained with the Deep Learning architecture.

All the scores here and in the appendix are numbers between 0 and 1, with 1 being the optimal performance. **We will show all the measures multiplied by 100,** since they are usually published in this format in the literature and it makes them easier to read.

5.2.1 Dummy baseline

We show in table 5.1 the performance measures that would be obtained by a dummy classifier that only knows the prior class distribution of the train set. **Recall that the train, validation and test sets are stratified by class labels, so the prior distributions coincide.** We also consider the subset of news with tweets, which is used by the Deep Learning architecture.

		Real news (%)	Fake news (%)	Dummy F1 score	Dummy accuracy
GossipCop	All news	75.8	24.2	38.9	75.8
	With tweets	74.6	25.4	40.5	74.6
PolitiFact	All news	53.7	46.3	63.2	53.7
	With tweets	46.6	53.4	69.6	53.4

Table 5.1: Dummy performance baseline.

Note that, since the F1 score depends on the sensibility, this measure is maximized when all news are predicted as fake. Hence, the optimal F1 score is the harmonic mean of the fake class prevalence and 1. Therefore, the higher the prevalence of fake news, the higher the F1 score baseline. On the other hand, the maximum accuracy is obtained when the dummy model assigns all news to the majority class, and therefore coincides with the prevalence of the greater class.

We must keep these baselines in mind when discussing the results. Observe that in the PolitiFact collection, which is almost balanced, the dummy F1 score baseline is 69.2 when using only news with tweets, which certainly would seem high in other situations.

Moreover, our PolitiFact dummy scores are even better than the results obtained by the FakeNewsNet dataset authors in [Shu+18] on trained models. However, in [Del+18], the authors reported F1 scores of 90 and more, which are coherent with our dummy scores and with the results that we present in this chapter. Therefore, **either the original dataset suffered important changes when it was published, or the results in [Shu+18] were not correctly reported.** As a consequence, the results shown in this chapter are not directly comparable to the scores reported in [Shu+18], although they seem to be in line with the results in [Del+18].

5.2.2 Non Deep Learning

Firstly, we show the results using all news, and then using only news with tweets. Recall from section 4.1.1 that numerical features store the number of images, movies and tweets of each news, categorical features contain information from the URLs of the domain and textual features are the title, text and author description of the news articles.

Using all news

We can see in figure 5.1 the test F1 scores of non Deep Learning algorithms using all features (remember that we repeat each combination 5 times). We highlight the dummy F1 baselines shown in table 5.1 with a red dashed line.

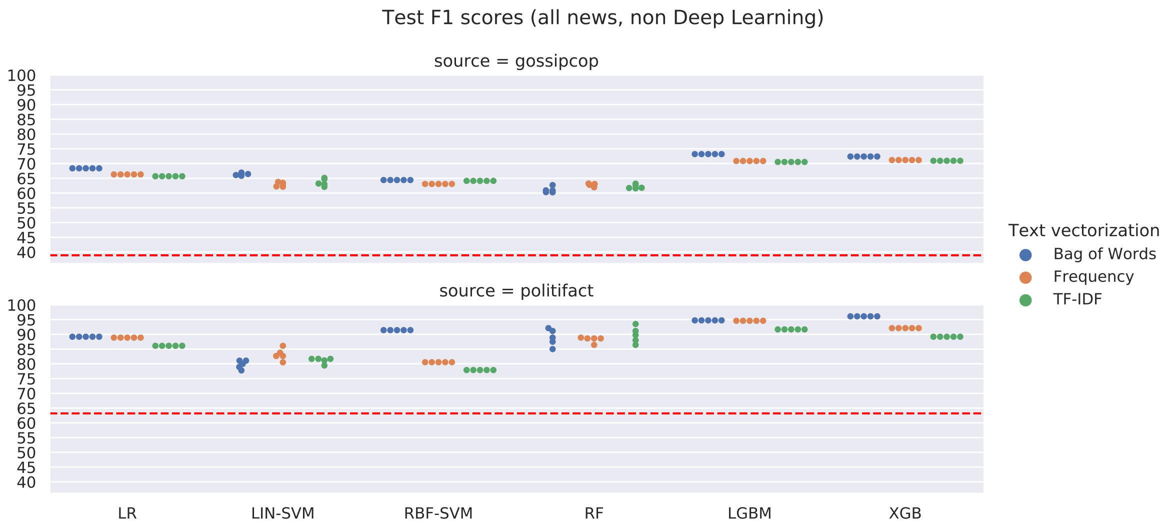


Figure 5.1: Test F1 scores by text vectorization and algorithm (all news, non DL vs DL).

There are quite a few interesting points to discuss:

- LightGBM and XGBoost achieve the best performance. The other algorithms also perform reasonably, all of them being clearly above the baselines. In the GossipCop collection, logistic regression leads SVM and random forest, although their performance is very similar. With PolitiFact news, however, random forest is slightly better than logistic regression, and SVM performs really well with the radial-basis-function kernel and bag-of-words Vector Space Model.
- The bag-of-words representation performs better than frequency count and TF-IDF, with most algoritms. The differences are greater in the PolitiFact collection, especially when using radial-basis-function SVM and XGBoost. TF-IDF obtains lower scores in most cases.
- Note the high variance of random forest F1 scores in the PolitiFact collection. Recall that random forest selects only a portion of features for each fitted tree.

Overall, we obtain reasonable results with the proposed algorithms. However, SVM models might need a bit of hyperparameter tuning, and it is also likely that the results obtained by RBF SVM with bag-of-words representations can be replicated by the other SVM models after a hyperparameter search.

Observe that our test F1 scores in the GossipCop collection are a bit better than the performance obtained by FakeNewsNet authors in [Shu+18], (they reported an F1 score of 59.5 for SVM and 64.6 for logistic regression). However, in the PolitiFact collection, our results are much better than in FakeNewsNet article (they reported F1 scores of 65.9 for SVM and 63.3 for logistic regression). As we explained above, other authors [Del+18] report some results similar to ours, so these differences might be due to a change in the dataset. **We remark once more that the scores presented here were obtained in the test set.**

We show in table 5.2 the average test F1 scores by text vectorization technique and algorithm, as in figure 5.1. We highlight the best score within each set of news.

	Algorithm	LR	LIN-SVM	RBF-SVM	RF	LGBM	XGB
Source	Vectorization						
GossipCop	Bag of Words	68.4	66.3	64.4	61.0	**73.2**	72.4
	Frequency	66.3	62.9	63.1	62.7	70.9	71.2
	TF-IDF	65.7	63.6	64.1	62.2	70.5	70.9
PolitiFact	Bag of Words	89.2	79.8	91.4	88.9	94.7	**96.1**
	Frequency	88.9	83.1	80.6	88.2	94.6	92.1
	TF-IDF	86.1	81.1	77.9	89.8	91.7	89.2

Table 5.2: Average test F1 score by text vectorization and algorithm (all news, non DL vs DL).

Let us now compare the usefulness of each type of feature. We can see in figure 5.2 the test F1 scores by algorithm and feature type. We show only the scores when textual features are encoded with the bag-of-words Vector Space Model, since we saw in figure 5.1 that it performs better and more consistently.

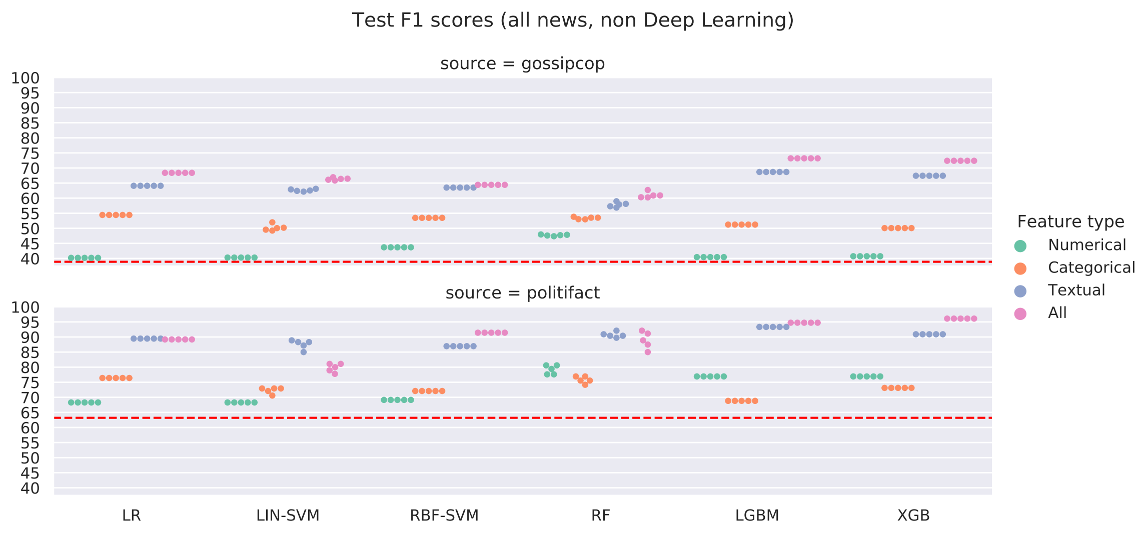


Figure 5.2: Test F1 scores by feature type and algorithm (all news, non DL vs DL).

We see that **numerical and categorical features are much less informative than textual features, but the performance using all features is better than using only textual features.** FakeNewsNet authors only used textual features in their non Deep Learning models, so this could be **another factor that explains the performance difference between our results and theirs.**

As an interesting detail, in the PolitiFact collection, SVM with linear kernel performs better using only textual features than using all features, and its performance is on par with with radial-basis-function SVM. It is also worth mentioning that, in the GossipCop collection, using only numerical features gives almost no information to the models, except when using a random forest.

We summarize the average test F1 scores by feature type in table 5.3.

Source	Algorithm / Feature type	LR	LIN-SVM	RBF-SVM	RF	LGBM	XGB
GossipCop	Numerical	40.2	40.3	43.7	47.7	40.5	40.7
	Categorical	54.4	50.2	53.5	53.4	51.2	50.1
	Textual	64.1	62.6	63.5	57.8	68.7	67.4
	All	68.4	66.3	64.4	61.0	**73.2**	72.4
PolitiFact	Numerical	68.3	68.3	69.1	79.2	76.9	76.9
	Categorical	76.4	72.3	72.1	75.8	68.8	73.1
	Textual	89.5	87.5	87.0	90.7	93.3	90.9
	All	89.2	79.8	91.4	88.9	94.7	**96.1**

Table 5.3: Average test F1 score by feature type and algorithm (all news, non DL vs DL).

Using news with tweets

We repeat the same analysis but using only news that have associated tweets. We can see in figure 5.3 that the test F1 scores when using only news with tweets are similar to using all news. However, we must keep in mind that, as we showed in table 5.1, the F1 score baselines are also higher than when using all news.

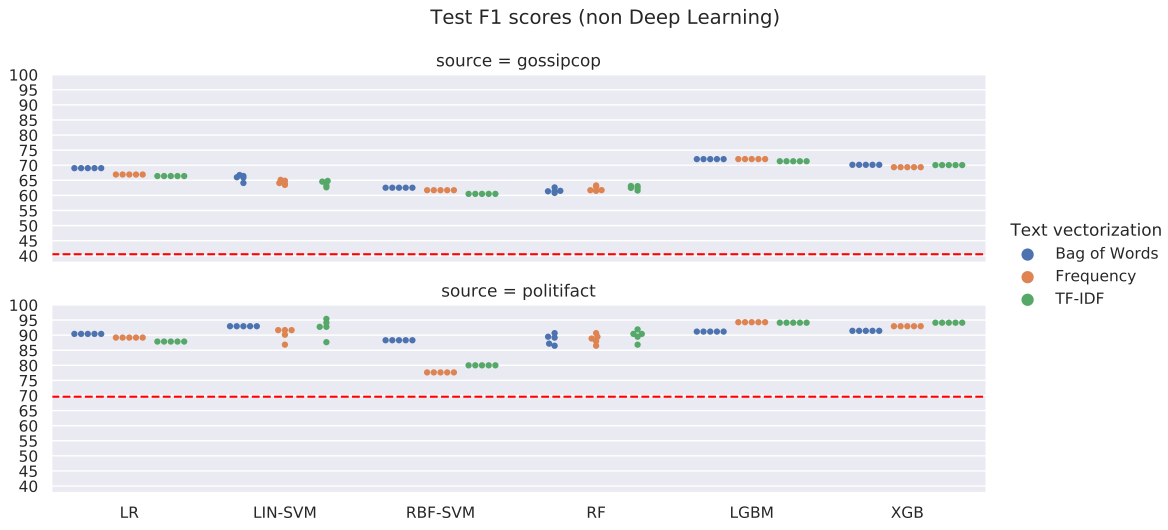


Figure 5.3: Test F1 scores by text vectorization and algorithm (non DL vs DL).

The remarks made when using all news are still valid, although we notice the following differences:

- In the GossipCop collection, F1 scores are similar compared to using all news, but there is a slight decrease in performance in all models.
- In the PolitiFact collection, SVM with linear kernels have improved greatly, from F1 scores in the 80-82 range to 90-92. We also notice a performance decrease in LightGBM and XGBoost. Furthermore, the bag-of-words representation technique has seen a small reduction in performance, while TF-IDF has improved a bit, especially in XGBoost, and frequency count does not change.
- It seems that **news with tweets are more challenging,** and that is why the bag-of-words Vector Space Model is most affected, since it gives less information. Also, **simpler models like logistic regression have smaller performance decreases,** which indicates that more complex models like XGBoost are overfitting the train set.

We gather in table 5.4 the average test F1 scores by text vectorization and algorithm, shown in figure 5.3.

Source	Algorithm / Vectorization	LR	LIN-SVM	RBF-SVM	RF	LGBM	XGB
GossipCop	Bag of Words	69.0	65.8	62.5	61.5	**72.0**	70.1
	Frequency	66.9	64.3	61.7	62.2	**72.0**	69.3
	TF-IDF	66.4	63.9	60.5	62.5	71.3	70.0
PolitiFact	Bag of Words	90.4	93.0	88.3	88.6	91.2	91.4
	Frequency	89.2	90.4	77.6	88.7	**94.3**	93.0
	TF-IDF	87.9	92.5	80.0	89.8	94.1	94.1

Table 5.4: Average test F1 score by text vectorization and algorithm (non DL vs DL).

Finally, we show in figure 5.4 the test F1 scores by feature type, when using only news with tweets. As we did earlier, we only present the results when textual features are encoded using the bag-of-words model, since it obtains good results and is consistent across most combinations.

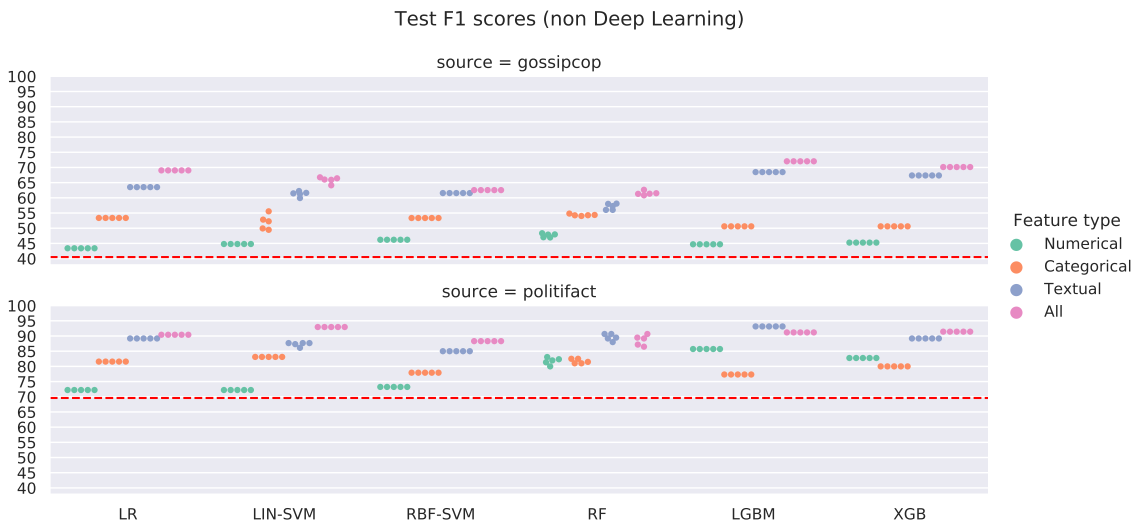


Figure 5.4: Test F1 scores by feature type and algorithm (non DL vs DL).

Figure 5.4 shows a picture similar to figure 5.2. However, there are some differences:

- In the GossipCop collection, these results are very similar with respect to using all news, but we notice a small drop in all models. Categorical features are less informative now, and numerical features are still of little use except with radial-basis-function SVM and random forest.
- Within PolitiFact, we notice an increase in performance when using textual or all features, and a decrease when using numerical or categorical features. Logistic regression and both SVM obtain F1 scores close to the dummy baseline when using numerical features, and categorical features are less informative with SVM and LightGBM. SVM with linear kernel does now obtain proper results when using all features.

We collect in table 5.5 the average test F1 scores by feature type and algorithm, presented in figure 5.4.

Source	Algorithm Feature type	LR	LIN-SVM	RBF-SVM	RF	LGBM	XGB
GossipCop	Numerical	43.4	44.8	46.2	47.6	44.7	45.3
	Categorical	53.4	52.0	53.4	54.3	50.6	50.6
	Textual	63.5	61.3	61.6	57.1	68.5	67.4
	All	69.0	65.8	62.5	61.5	**72.0**	70.1
PolitiFact	Numerical	72.2	72.2	73.2	81.8	85.7	82.8
	Categorical	81.6	83.1	77.9	81.7	77.3	80.0
	Textual	89.2	87.3	85.0	89.6	**93.2**	89.2
	All	90.4	93.0	88.3	88.6	91.2	91.4

Table 5.5: Average test F1 score by feature type and algorithm (non DL vs DL).

5.2.3 Deep Learning

We will now present the main results of the Deep Learning architecture, and see if using information from tweets and user profiles helps improve performance. As with the non Deep Learning architecture, we will firstly show a comparison by text vectorization, and then by feature type.

Figure 5.5 shows the test F1 scores by text vectorization technique and features used. We can quickly see that **the performance in the GossipCop collection improves greatly when adding tweet and user profiles information,** compared to using only news content. It also improves a bit in the PolitiFact colletion, but the F1 scores were already very high.

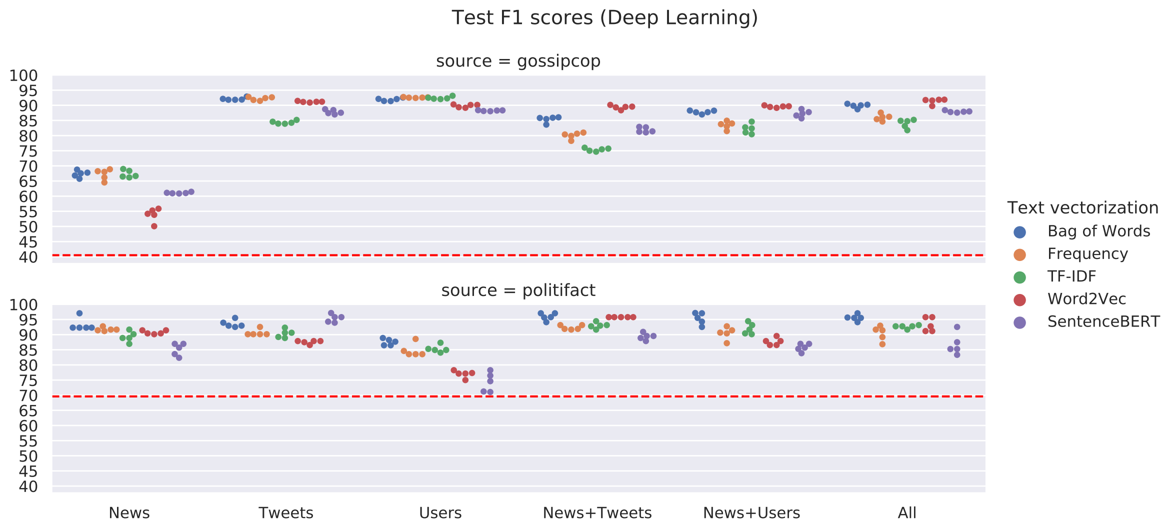


Figure 5.5: Test F1 scores by text vectorization and feature origin (DL).

We notice some important aspects that are worth mentioning:

- Tweets and user profiles contain a lot of information, especially in the GossipCop set of news. We see that using only tweets or user profiles is much more informative than using the news themselves. This has to be related with what we saw in the exploratory data analysis (see section 3.6): many user accounts that posted comments about real news still had the default profile options. We also observed a large number of tweets whose source was related to automation tools like IFTTT or dlvr.it. Our hypothesis is that:
  - When using only tweet features, our architecture is able to easily detect if the news are real by looking at the tweet source. If tweets source is IFTTT or dlvr.it, they are more likely

related to real news.

  - When using only user profiles, the model looks especially at the user profile customization options, and if users have not changed the default options, news are more likely to be real.

  Therefore, **our architecture might actually be exploiting the fact that GossipCop real news seem to use bot accounts,** which is *unexpectedly* not as common within fake news, and therefore is able to detect real news rather than fake news.

- PolitiFact news, however, did not show such extreme particularities in the exploratory data analysis, and that might be why using only user profile information gives worse F1 scores. Therefore, it might be the case that, in general, **user information is very useful, but not as useful as it shows in the GossipCop collection.**

- Using only tweet features gives very good results in both collections, even though PolitiFact-related tweets were also not showing the particularities observed in GossipCop tweets, like tweet sources related to automation tools. This means that, even without *strange* phenomena like bot tweets seemingly more related to real news than fake news, **tweets contain a lot of information useful to detect fake news, maybe more than user profiles.**

- It is interesting to see that, in the GossipCop collection, **mixing news features with tweets or user profiles decreases the model performance,** especially when using Vector Space Models. However, if we mix all three, F1 scores raise again.

- Regarding textual representations, we see a high variability depending on the chosen technique. Overall, **Vector Space Models seem to perform very well in all the combinations, although bag-of-words has a small advantage** over frequency count and TF-IDF. On the other hand, Word2Vec struggles a bit with GossipCop using only news and PolitiFact using only user profiles, but it performs well in both collections when using all features. SentenceBERT is a bit behind in most combinations, although it excels in PolitiFact tweets, and performs well in GossipCop with all features. All things considered, **Word2Vec seems to be the strongest technique if all features are used.**

We expected to obtain better results using Word2Vec and, especially, SentenceBERT. It seems that Vector Space Models perform well because it is more important to detect certain words than to understand the meaning of each sentence. It could also happen that this dataset is biased in some sense, and the performance in other datasets is actually lower, especially considering that automation tools and bot accounts seem to be more common in real news than in fake news. **Keep in mind that the test set is a representative set of the whole dataset, but the FakeNewsNet dataset itself might not be a representative subset of the whole set of news and their user engagements.** In other words, it is possible to obtain really good F1 scores in an independent test set and, at the same time, obtain worse results in other datasets.

We have identified some possible explanations on why the performance of Word2Vec and SentenceBERT is lower than ideal:

- **Word2Vec has its own vocabulary set, and some tokens used by our Vector Space Models are not in its vocabulary.** Therefore, our architecture cannot use that information, and that might lead to worse performance. However, it is striking that Word2Vec performs so badly when using only GossipCop news, since news articles should contain common tokens. One solution could be to use the same tokenization than Word2Vec. Another would be to raise the vocabulary size.

- **SentenceBERT might need to be fine-tuned** to unlock its potential. Recall from section 2.2 that the original BERT was designed to be fine-tuned for each specific task and, although SentenceBERT was specifically fine-tuned to obtain sentence representations, it might be necessary to fine-tune SentenceBERT on a subset of news before using it.

We show in table 5.6 the mean test F1 scores from figure 5.5. The best scores for each source and column are highlighted.

Source	Feature origin Vectorization	News	Tweets	Users	News+Tweets	News+Users	All
GossipCop	Bag of Words	**67.3**	92.1	91.9	85.4	87.8	89.8
	Frequency	67.2	**92.2**	**92.6**	80.0	83.5	86.0
	TF-IDF	67.3	84.4	92.4	75.4	82.2	84.0
	Word2Vec	53.8	91.2	89.8	**89.4**	**89.6**	**91.4**
	SentenceBERT	61.1	87.8	88.2	81.9	87.2	87.9
PolitiFact	Bag of Words	**93.3**	93.6	**87.6**	95.9	**95.3**	**95.6**
	Frequency	91.7	90.6	84.8	92.4	90.5	90.4
	TF-IDF	89.3	90.3	85.3	93.0	92.0	92.6
	Word2Vec	90.8	87.5	77.0	**95.8**	87.7	93.3
	SentenceBERT	85.1	**95.4**	74.3	89.4	85.8	86.8

Table 5.6: Average test F1 score by text vectorization and feature origin (DL).

We can compare the first column of table 5.6, which contains the results when only news features are used, with the results of non Deep Learning algorithms in table 5.4. We see that our Deep Learning architecture performs roughly at the same level as the proposed non Deep Learning models (and well above the F1 score baselines), although a bit worse than the best-performing LightGBM algorithm (F1 score of 72.0 on the GossipCop collection, 94.3 on the PolitiFact collection). All the Vector Space Models perform equally well using the Deep Learning architecture, but TF-IDF performs a bit worse than bag-of-words or frequency count in the PolitiFact source, following the remarks made in subsection 5.2.2. However, it is clear that the non Deep Learning architecture cannot compete when we add tweet and user information to our Deep Learning model.

Regarding the usefulness of each type of feature, figure 5.6 shows the test F1 scores by feature type and feature origin. Textual features have been represented using the bag-of-words Vector Space Model, since it was the most consistent technique.

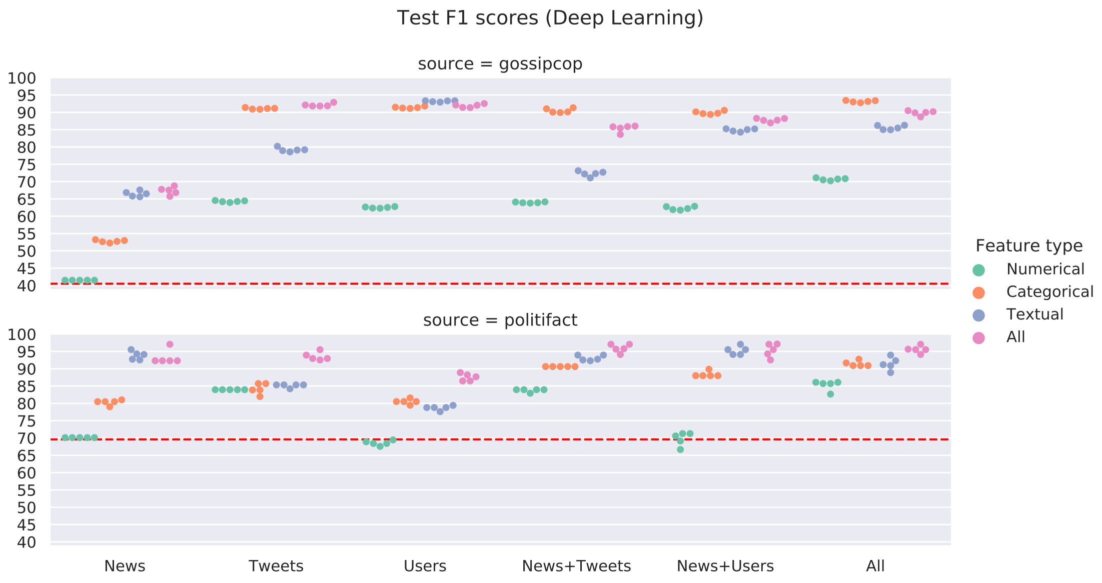


Figure 5.6: Test F1 scores by feature origin and feature type (DL).

We can draw some conclusions from this chart, in line with what we have already discussed:

- **In the GossipCop collection, categorical features from tweets and user profiles obtain very high F1 scores.** Note that the tweet source and most profile customization features are categorical and, as we explained, there is a connection between these features and the class labels. Also, **numerical features from tweets and user profiles seem to be very informative, especially when using both tweets and user profiles** (number of likes, retweets, friends, followers, delay between tweets, time from user creation to tweet). Regarding textual features, **the user description obtains the highest F1 score,** much higher than the tweet texts.

- **Within the PolitiFact collection, news textual features are the most important features,** and it seems that user descriptions and tweet texts are much less informative. Numerical features from news or user profiles are almost useless (recall the dummy F1 score is 69.6), but those from tweets obtain higher F1 scores. Categorical features do not have the same power as in the GossipCop collection, but they are not useless either.

- An important remark is that, in the GossipCop collection, there are three combinations leading the results (tweets categorical, users categorical and users numerical), while in the PolitiFact collection, the effect of the different feature types are additive. Note how using all the features from tweets or user profiles gives much better results than each type of feature alone. **This might indicate that the F1 scores obtained in the GossipCop collection are exceptional, and are due to the particularities already highlighted** (bot or automated accounts used massively by real news). We cannot reject this hypothesis using the PolitiFact set either, since the scores are very high even when using non Deep Learning algorithms, so we would have to test our Deep Learning architecture with other datasets.

Finally, we show in table 5.7 the mean test F1 scores from figure 5.6.

	Feature origin	News	Tweets	Users	News+Tweets	News+Users	All
Source	Feature type						
GossipCop	Numerical	41.5	64.3	62.5	64.0	62.3	70.7
	Categorical	52.8	91.1	91.4	**90.5**	**89.9**	**93.2**
	Textual	66.5	79.2	**93.2**	72.3	84.9	85.6
	All	**67.3**	**92.1**	91.9	85.4	87.8	89.8
PolitiFact	Numerical	70.1	84.0	68.5	83.7	69.8	85.2
	Categorical	80.3	84.2	80.5	90.6	88.4	91.4
	Textual	**93.8**	85.1	78.7	93.1	**95.3**	91.4
	All	93.3	**93.6**	**87.6**	**95.9**	**95.3**	**95.6**

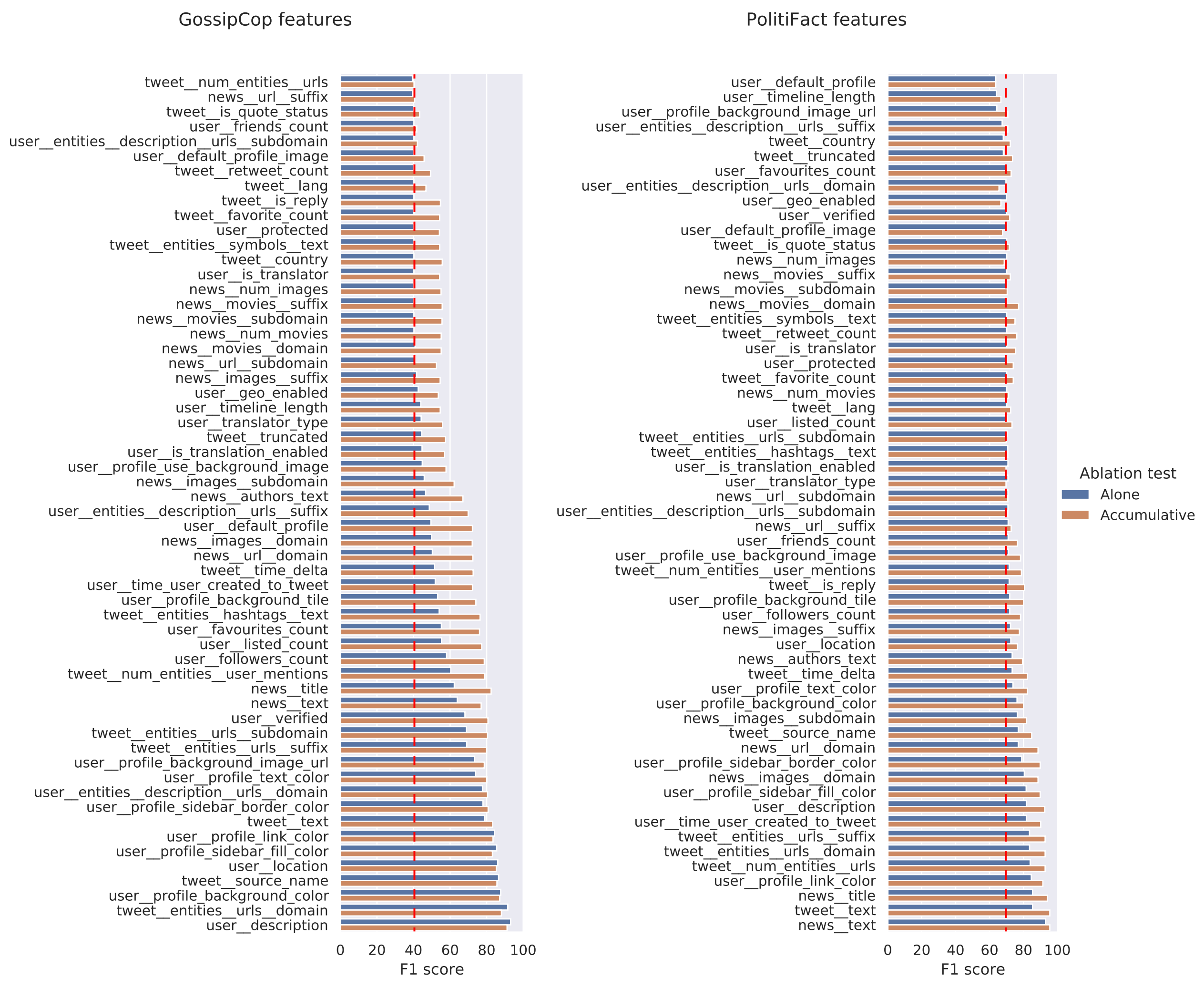
Table 5.7: Average test F1 score by feature origin and feature type (DL).

If we compare the results of this architecture using only news features (first column) with the results of non Deep Learning models (shown in table 5.5), we see that the remarks in 5.2.2 also apply to the Deep Learning architecture. Using only numerical features gives F1 scores close to the dummy baseline (40.5 in GossipCop, 69.6 for PolitiFact), while using categorical features gives better results, in the range of non Deep Learning algorithms. It is worth mentioning that in the PolitiFact collection, the best F1 score is achieved when only textual features are used, and it is better than the 93.2 points obtained by LightGBM.

5.2.4 Deep Learning Ablation test

Finally, we present the results of the ablation tests using the Deep Learning architecture. GossipCop and PolitiFact news are analyzed separately, following the methodology explained in section 5.1.2. In this case, we show the results when textual features are encoded using Word2Vec, because we saw in figure 5.5 that it performs better than the other techniques when using all features on the GossipCop collection, and excels when we mix news text with tweet and user information, meaning that it is the most versatile representation technique.

Figure 5.7 shows the test F1 scores for each source, with features ranked in ascending order. Blue bars indicate the test F1 score for each feature alone, while orange bars represent the F1 score using that feature and all above. We highlight the dummy F1 score baselines (see table 5.1) with a red dashed line.



(a) GossipCop. (b) PolitiFact.

Figure 5.7: Ablation test F1 scores (DL). Textual features are vectorized with Word2Vec.

Let us analyze the ablation test results within each news source:

- **GossipCop.** We see that all the features up to `news__num_movies` obtain F1 scores similar to the dummy baseline (40.5), which means that they carry almost no information to detect fake news. Some of these features are the number of favorites and retweets of each tweet, the tweet language (most tweets are in english) or the number of friends of each user. On the other hand, features below `news__title` obtain F1 scores greater than 60. This group includes features like news text and several profile customization features. Finally, features below `tweet__text` obtain F1 scores higher than 80. All of them are profile customizations, except the tweet source name and the domain of tweet entities.

  Overall, we see that the most performing features are the user description (textual), user profile customization categories (categorical) and the tweet source name (categorical). Then, we drop down to news text and title (textual). This is coherent with figure 5.6, where we saw that tweet categorical features were very powerful, and user profile categorical and textual features too. This reinforces the hypothesis that our architecture is exploiting the connection between real news and user accounts with little profile customization. Note that the accumulative models perform better than each individual feature until the tweet text. The remaining features perform better alone than with all the previous features, which indicates that those features are exploiting that connection

- **PolitiFact.** Most features have F1 scores a bit better than the dummy baseline (69.6). Note that the highest F1 score is obtained by `news__text`, with a score close to 90. As we saw in figure 5.6, the performance was very high just using textual features from news, so it is not surprising to see that the news text obtains such a high F1 score. However, contrary to the GossipCop collection, within the features with highest F1 score, the accumulative models perform better than each feature alone. We see that the user description is not as informative as in the GossipCop collection, and the profile customization features are in the 70-80 range. Numerical features like the time from user creation to the tweet publish date obtain higher performance than in GossipCop.

Chapter 6

Conclusions and Future work

In this work, we have tested various fake news detection systems, both based on Deep Learning and on several well-known algorithms, on two sets of news about different topics. We have tried using only news content and including user engagements. Moreover, we have tested different textual representations, based on Vector Space Models, Word2Vec and SentenceBERT. We have also tried using different subsets of features, including an ablation test on each set of news.

We have found that **adding information from user engagements greatly improves the results with respect to using only news content.** In the GossipCop collection, features related to user profile customization contributed most to the final performance, especially the user description, with some tweet features like the URLs linked and the tweet source being very relevant. With PolitiFact news, the most relevant feature was the news text, although user engagement features like tweet text or user customization were still important and contributed to the final model.

These results agree with our findings in the exploratory data analysis, which shows that GossipCop real news must be using automated tools or bot accounts to publicize their news, rather than fake news. We also found that this was connected to a high increase in the number of users with the default profile options. Our feature correlation study confirmed what we found in the data analysis and also revealed that features like the news publishing media or the number of tweet user mentions and links are correlated with news labels.

Regarding these particularities of FakeNewsNet, it might be the case that FakeNewsNet is not a representative dataset of the whole set of news, tweets and users. After all, the test split on which we evaluate our models is a representative subset of FakeNewsNet, but if FakeNewsNet is not representative of the real set of existing news and user engagements, it is possible to obtain really good F1 scores in the test set and, at the same time, obtain worse performance in other datasets.

However, even if the effect of adding user engagements is overestimated in the GossipCop collection, we can see in the ablation test results (see figure 5.7) that some user profile features are also among the most relevant in the PolitiFact collection. Moreover, observe in figure 5.6 that all types of features contribute equally in this collection when using only tweet information, and perform even better when combined, as opposed to the GossipCop collection, where there is a leading type of feature. This suggests that there is indeed an improvement when user engagements are included in the model, although perhaps smaller than observed in our tests.

It is worth mentioning that our results in the PolitiFact collection are similar to those in [Del+18]. Recall that those results did not coincide with the results reported in the original article [Shu+18]. We believe that either the publicly available FakeNewsNet dataset is different from the dataset used in [Shu+18] or the dataset authors tested their models under different conditions.

Regarding textual representation techniques, our experiments show that **Vector Space Models outperformed Word2Vec and SentenceBERT, especially the bag-of-words model.** As we discussed in section 5.2, this means that it is more important to detect certain words than to understand the meaning of sentences, as it happens in tasks like spam detection. However, Word2Vec performed best in the GossipCop collection when using all features, suggesting that is the most versatile model. And SentenceBERT performed best when using PolitiFact tweets, suggesting that understanding the meaning is more important in some topics. Overall, according to our results, the best option is to use the bag-of-words model, Word2Vec or a fine-tuned SentenceBERT.

Our results also prove that **different types of features perform better in different parts of the dataset.** We found that numerical features from news gave poor results, but numerical features from tweets and users gave results comparable to using all the information from news, especially in the GossipCop collection. We saw in the ablation test that features like the number of mentions in a tweet, the number of followers of a user or the number of links in the tweet were among the most relevant features.

Looking at categorical features, our results say that they are very relevant in the GossipCop collection, since they contain the information related to the tweet source and user profile customization, both particularly characteristic of GossipCop news. These features are still quite relevant within PolitiFact news, but they do not perform so well.

Concerning textual features, the most important text features in the GossipCop collection were the user description and the tweet text, while in the PolitiFact set, news text was the most informative. The content of GossipCop articles gave worse results than the text from related tweets or the user descriptions, contrary to what we observed in PolitiFact news. This suggests that the performance of content-based models might be strongly dependent on the topic.

If we look at our results using non Deep Learning algorithms, we see that LightGBM performed better than the other algorithms, and even a bit better than our Deep Learning architecture using only news features. XGBoost was closely behind, and logistic regression also obtained good results. We noticed that SVM models might need hyperparameter tuning, and random forest results performed a bit worse than the rest, perhaps because each tree is fitted with a random subset of features, and we have explained that detecting certain words is key to obtain better results. This might be the same reason that simpler representation techniques like bag-of-words VSM perform so well.

To finish our work, we propose some **future work that might be worth exploring.** Firstly, it would be interesting to test our architecture in other datasets with different label granularity, news from other sources, topics and languages, and user engagements from other platforms. This could also require to make some changes in the architecture, like using a multilingual textual representation technique (there are multilingual BERT models, for instance) and standardizing the set of features from user engagements so that all of them are available from the APIs of all platforms involved.

Another line of work could be to train fake news detection models using a train set composed of news from one source (or topic) and evaluating on news from other source, and study the performance impact and which features are more relevant for each source. This should tell how generalizable the model is.

Finally, we believe that it could be useful to test our architecture in the context of early fake news detection, including only user engagements up to a short period of time since news were published. It might be possible to build a fake news detector capable of extracting information from short-term user engagements more efficiently, so that it only needs to see the initial reaction to distinguish between fake and real news.

Bibliography

[Ped+11] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[Mik+13] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: `1301.3781 [cs.CL]`.

[PSM14] Jeffrey Pennington, Richard Socher, and Christopher D Manning. “Glove: Global vectors for word representation”. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[MG15] Tanushree Mitra and Eric Gilbert. “Credbank: A large-scale social media corpus with associated credibility annotations”. In: *Proceedings of the International AAAI Conference on Web and Social Media*. Vol. 9. 1. 2015. URL: `https://github.com/compsocial/CREDBANK-data`.

[Del+16] Michela Del Vicario et al. “The spreading of misinformation online”. In: *Proceedings of the National Academy of Sciences* 113.3 (2016), pp. 554–559.

[Sil+16] Craig Silverman et al. “Hyperpartisan Facebook pages are publishing false and misleading information at an alarming rate”. In: *Buzzfeed News* 20 (2016), p. 68. URL: `https://github.com/BuzzFeedNews/2016-10-facebook-fact-check`.

[HA17] Benjamin Horne and Sibel Adali. “This just in: Fake news packs a lot in title, uses simpler, repetitive content in text body, more similar to satire than real news”. In: *Proceedings of the International AAAI Conference on Web and Social Media*. Vol. 11. 1. 2017.

[ÖG17] Özlem Özgöbek and Jon Atle Gulla. “Towards an understanding of fake news”. In: *CEUR workshop proceedings*. Vol. 2041. 2017, pp. 35–42.

[Pot+17] Martin Potthast et al. “A stylometric inquiry into hyperpartisan and fake news”. In: *arXiv preprint arXiv:1702.05638* (2017).

[RSL17] Natali Ruchansky, Sungyong Seo, and Yan Liu. “Csi: A hybrid deep model for fake news detection”. In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 2017, pp. 797–806.

[Tac+17] Eugenio Tacchini et al. “Some like it hoax: Automated fake news detection in social networks”. In: *arXiv preprint arXiv:1704.07506* (2017). URL: `https://github.com/gabll/some-like-it-hoax`.

[Vas+17] Ashish Vaswani et al. “Attention is all you need”. In: *arXiv preprint arXiv:1706.03762* (2017).

[Del+18] Marco L Della Vedova et al. “Automatic online fake news detection combining content and social signals”. In: *2018 22nd conference of open innovations association (FRUCT)*. IEEE. 2018, pp. 272–279.

[Dev+18] Jacob Devlin et al. “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805* (2018). URL: `https://github.com/google-research/bert`.

[SW18] Giovanni Santia and Jake Williams. “BuzzFace: A News Veracity Dataset with Facebook User Commentary and Egos”. In: *Proceedings of the International AAAI Conference on Web and Social Media* 12.1 (June 2018), pp. 531–540. URL: `https://github.com/gsantia/BuzzFace`.

[Shu+18] Kai Shu, Deepak Mahudeswaran, Suhang Wang, et al. “FakeNewsNet: A Data Repository with News Content, Social Context and Dynamic Information for Studying Fake News on Social Media”. In: *arXiv preprint arXiv:1809.01286* (2018). URL: `https://github.com/KaiDMML/FakeNewsNet`.

[Zha+18] Jiawei Zhang et al. “Fake news detection with deep diffusive network model”. In: *arXiv preprint arXiv:1805.08751* (2018).

[BS19] Pranav Bharadwaj and Zongru Shao. “Fake news detection with semantic features and text mining”. In: *International Journal on Natural Language Computing (IJNLC) Vol* 8 (2019).

[RG19] Nils Reimers and Iryna Gurevych. “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Nov. 2019. URL: `http://arxiv.org/abs/1908.10084`.

[SML19] Kai Shu, Deepak Mahudeswaran, and Huan Liu. “FakeNewsTracker: a tool for fake news collection, detection, and visualization”. In: *Computational and Mathematical Organization Theory* 25.1 (2019), pp. 60–71.

[Wol+20] Thomas Wolf et al. “Transformers: State-of-the-Art Natural Language Processing”. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. URL: `https://www.aclweb.org/anthology/2020.emnlp-demos.6`.

[KGN21] Rohit Kumar Kaliyar, Anurag Goswami, and Pratik Narang. “FakeBERT: Fake news detection in social media with a BERT-based deep learning approach”. In: *Multimedia Tools and Applications* 80.8 (2021), pp. 11765–11788.

Appendix A

Architecture implementation

This appendix contains the implementation of the architectures described in chapter 4. The non Deep Learning architecture is built mainly using *Scikit-learn*, while the Deep Learning architecture is a *Tensorflow* Functional model. We describe the former in section A.1 and the latter in A.2.

A.1 Non Deep Learning

The non Deep Learning architecture is built upon the *Scikit-learn* framework. Based on this framework, we use the classification algorithms described in section 4.1 to have a baseline to which we can compare other models later on.

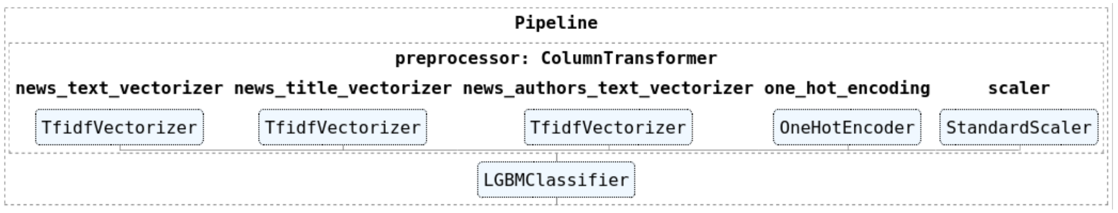


Figure A.1: Scikit-Learn model example.

The general structure of each model is a *Pipeline* object that performs two steps. This structure takes advantage of the tools within Scikit-learn and makes the testing process a bit easier.

1. **Preprocessing and text vectorization.** A *ColumnTransformer* object that applies the corresponding preprocessing to each column. This object allows to apply different preprocessing steps to different sets of columns. The steps are specified as a list of tuples, with each tuple containing the name of the step, the step to apply and the affected columns.

   - Numerical features are centered and scaled using *StandardScaler*. We can directly pass all the numerical features to the same StandardScaler and it will automatically handle each feature independently.
   - Categorical features are encoded using *OneHotEncoder* with the parameter `handle_unknown='ignore'`, since we might come across with values in the test set that have not been seen in the train set. Again, we can pass all the categorical features to the same encoder and they will be treated independently.
   - Textual features are tokenized and vectorized using either *CountVectorizer* with parameter `binary=True` (bag of words) or `binary=False` (frequency counts), or *TfidfVectorizer* (TF-IDF). In all cases, we will use the `max_features=5000` parameter to limit the vocabulary to the 5000 most frequent tokens. Furthermore, each feature will have its own vectorizer, since the vocabulary is not be shared between features.

2. **Classification.** We use the following implementation of the classification algorithms:

    - **Logistic regression.** We use the *LogisticRegression* implementation of Scikit-learn with parameter `max_iter=500`, since the algorihtm would sometimes not converge with the default 100 iterations.
    - **Support vector machines.** We use *LinearSVC* for the linear kernel SVM algorithm and *SVC* for the radial basis function kernel. Both are implemented in Scikit-learn.
    - **Random forest.** Implemented in Scikit-learn as *RandomForestClassifier.*
    - **LightGBM.** Using the official `lightgmb` Python package, which includes the *LGBMClassifier* class following Scikit-learn's design.
    - **XGBoost.** Similarly, we use the *XGBClassifier* estimator contained in the official `xgboost` package.

Models are fed from two different *pandas's dataframes*: one containing training news (when training) and another with test news (when evaluating). These dataframes were stored at the end of section 3.5, from the training and test splits, respectively. Categorical features are stored as pandas categorical type, and we checked that there is no data leaking from the test set, so that the model has no information on possible values that are not in the train set.

A.2 Deep Learning

The Deep Learning architecture is implemented using Tensorflow's Functional API and Sequential API. We take advantage of Tensorflow's features like the Dataset API and ragged tensors, which are crucial to handle the news-tweets one-to-many relationship.

This section is structured as follows: we explain in subsection A.2.1 how we use Tensorflow's Dataset API and ragged tensors that will serve to feed the models both in training and evaluation. Then, we describe in A.2.2 the construction of the Functional model, specifying which layers are used in each part. Finally, we explain in subsection A.2.3 the modifications we had to make to use SentenceBERT, since BERT models are very resource-demanding and we could not embed SentenceBERT inside our Functional model.

A.2.1 Input Data. Datasets and ragged tensors

Each piece of news is associated to a variable number of tweets and each of the tweets is enriched by adding information about the user that posted the tweet. We join news and tweets using the news identifiers, and tweets and user profiles by the user's Twitter ID. The main problem we have to figure out is how to handle this *one-to-many* relationship between news and tweets. We have two options:

- Padding and truncating to a fixed number of tweets. We could pad all the news to have exactly 100 related tweets (or any other number, but as we explained earlier, we are limiting ourselves to 100 tweets). Then, we would *only* have to add dummy tweets that contain no information or truncate up to these many tweets.

- **Handle varying-length sequences of tweets.** This would natively accept any number of tweets for each piece of news and could even improve model's throughput, since no useless information is added.

Apart from this, we have another issue: we have to somehow join news, tweets and user profiles together *before* feeding the data to the model. This is, **we have to create a *dataset* containing *batches* of *samples*,** each sample being a piece of news along with the related tweets and users's information (from now on, whenever we mention the word *sample*, we will refer to this). Note that, at this point, the information is stored as separated *pandas*'s dataframes (stored after the train-validation-test split in section 3.5), and we have to end up with a joint dataset containing all the information for each sample. Furthermore, **our goal is to create this dataset in such a way that we can later choose which features we want to use,** since part of our tests will consist on comparing models that use different subsets of features.

After some research, we found that `tf.RaggedTensor` could be a solution to properly handle varying-length sequences, so we decided to explore this possibility and try to build the models without adding dummy tweets. A `RaggedTensor` is a `Tensor` that has at least one *uniform* dimension and one or more *ragged* dimensions. The outermost dimension is always uniform by definition, since a `Tensor` is a list of elements. Any other dimension whose size is always the same for each element is a uniform dimension, as opposed to ragged dimensions, whose size is variable. In our case, the ragged dimension will be the *tweet* dimension.

Once the first question is clear, we have to find a way to join news, tweets and user profiles to create a `tf.Dataset`. Again, after spending some time finding possible solutions, we decided to use `Dataset.from_generator` to **create a `Dataset` from a generator function that yields one sample at a time.** This function needs as arguments the generator function and the *output signature* of the dataset, which we explain below. However, using generators (which are actually the *bottleneck*) reduces the model's throughput, since it will be called many times to obtain the samples, and to obtain each sample we have to find which tweets are related to each piece of news. We will solve this problem by **saving the dataset to disk,** so that we only have to join news, tweets and user profiles once.

- **Generator.** To yield samples, we will simply iterate through the set of news, finding the tweets associated to each one of them. For each piece of news, the generator will find the tweets related (inner join with the news identifier), and then the user profiles of the tweet poster (inner join with the tweet ID), and will yield a dictionary with feature names as keys. News features' keys will store the value itself as the key's value, whereas tweet and user features will contain a one-dimensional `Tensor` with the values of that feature for each of the associated tweets. Outputting the samples as dictionary will allow us to easily handle the features by name instead of by index. Note that all the tweet and user features will contain the values in the same order as the tweets, despite following a column-style design.

  Note. From now on, our `Datasets` will not contain news, tweet or user identifiers, since they are no longer needed.

- **Output signature.** This argument describes the shape of the yielded elements. In our case, it will be a dictionary with feature names as keys and `TensorSpec` objects as values. Each `TensorSpec` needs to specify the shape and the `dtype` of the stored value. News features will have `shape=()`, since they store scalar values, while tweet and user features will have `shape=(None,)`, as they contain a variable-size one-dimensional `Tensor`. The `dtype` will always be either `tf.float32` (numerical features), `tf.string` (textual and categorical features) or `tf.int32` (target labels), following the type of feature as explained in tables 3.9, 3.10 and 3.11.

We create three separate `Datasets`, each one of them containing only samples from the training, validation and test sets of news, respectively. Then, we save the three `Datasets` to disk, using the experimental function `Dataset.save`. To load the `Datasets` later, we use `Dataset.load`, passing as argument the same output signature we used to create each `Dataset`.

Furthermore, note that due to the inner joins used in the generator, news without tweets will not be contained in the `Datasets`. **We will create three separate `Datasets` (train, validation, test) containing all news and only news' features,** following the same procedure described above but modifying the generator to only output news features. During the tests, **we will mainly focus on the `Datasets` that**

contain also tweets and user profiles, but these other `Datasets` will be useful to see whether the observed differences in the results come from using only news with tweets.

Note. We checked that each of these `Datasets` contain only news from the corresponding set of news (train, validation or test). We ensured this by having different generators instead of a function that takes a dataframe as argument, since the `Dataset` calls the generator when samples are required and this would not be ensured otherwise.

Preparing Datasets for training

At this point, we already have our `Datasets` full of samples, but we still have to apply some steps like separating the target label and optimizing for performance. We follow these steps in order:

- **Filter news source.** We are testing the model with news from each source separately. Therefore, we need to exclude news from the source we are not training on. We do this by using `Dataset.filter` combined with `tf.strings.regex_full_match` on the field `news__source`.
- **Set news label as target.** We remove the target label from the features and output each sample as a tuple `(features, label)`.
- **Make batches of samples.** Using `Dataset.dense_to_ragged_batch`, we split the set of samples in groups of *batch size* news and join them into dictionaries whose values are `Tensors` with an extra dimension. News features will be gathered in a one-dimensional `Tensor`, while tweet and user features will be transformed into a two-dimensional `RaggedTensor`, with the second dimension ragged. Each element of the batched `Dataset` will be a dictionary containing the features of as many news as the batch size, except the last batch, which might have less news.

  Note. We cannot use `Dataset.batch` because it does not support creating `RaggedTensors` yet.
- **Dataset optimizations.** Using `Dataset.cache` creates a cache that speeds up second and successive readings, while `Dataset.prefetch` preloads the next batch while the current batch is being processed.

A.2.2 Functional model

We use Tensorflow's *Functional* API to build the architecture. Most of the layers used come from `tf.keras`, although we also need some Tensorflow native layers. We first define the input layers, which depend on what features we want to use. Then we define the preprocessing for each feature, including the representation technique of textual features. Next, we explain how to *reduce* the tweets dimension, to finally pass the summarized vectors to a prediction head, which will output the probability of news being fake.

Input layers

For each feature we want to use, we create an `Input` layer with parameters:

- `name=<feature name>`.
- `dtype=<feature dtype>`, according to the feature type in tables 3.9, 3.10 and 3.11. Categorical features will be treated as strings, since Tensorflow does not support *pandas* categorical type.
- `shape=()` for news features, otherwise `shape=(None,)`.
- `ragged=False` for news features, else `ragged=True`.

We use the `name` property because this allows us to take the values by key from the dictionaries fed by the `Dataset`. This helps handling such a high number of features.

Preprocessing and textual representation

Each feature has a preprocessing sequence with one or more preprocessing layers, which we chain sequentially from the feature's input layer. All the layers that need to `adapt` (fit some parameters according to data) will be adapted independently and using only the training `Dataset`. All the layers used here will be set as non-trainable.

- Each numerical feature is centered and scaled using a `Normalization` layer.
- Each categorical feature is encoded by combining a `StringLookup` layer with a `tf.one_hot` operator. We will specify the `StringLookup` parameter `max_tokens=100` to limit to the 100 most frequent values, and in the one-hot layer we will specify the parameter `depth=<StringLookup vocabulary size>`, to match the lookup vocabulary size (might be lower than 100).
- Textual features are encoded using either a Vector Space Model (bag-of-words, frequency count or TF-IDF), or pretrained word embeddings (Word2Vec or SentenceBERT). Each feature will have its own chain of layers, which means that there is not a shared vocabulary, for instance. We can choose which technique to apply, but the same technique will be used for all the features.
  - **Bag of words.** We use a TextVectorization layer with parameters: `max_tokens=5000, output_mode=binary, pad_to_max_tokens=False`. With these settings, this layer will tokenize the text (lowercase, remove punctuation and split by whitespaces) and return the bag-of-words representation of the string. When calling the `adapt` method, it will create the vocabulary, retaining the 5000 most frequent tokens at most (some features might have less than 5000 different tokens). This vocabulary will then be used to compute the string representation when the layer is used in the model.
  - **Frequency count.** We use the same layer and arguments as in the bag-of-words model, but changing to `output_mode=count`, to return frequency counts instead of zeros and ones.
  - **TF-IDF.** Same as in the above cases but using `output_mode=tf-idf`.
  - **Word2Vec.** We use *gensim* to download the weights from Google's Word2Vec model (specifically, `word2vec-google-news-300`), decompress the file and load it using `gensim.models.KeyedVectors.load_Word2Vec_format`. Since there might be tokens that are in the feature's strings but not in Word2Vec's vocabulary, we first obtain the feature's vocabulary in the training `Dataset` and select the 5000 most frequent tokens that are also in Word2Vec's vocabulary. The preprocessing layers will consist on a `TextVectorization` layer to tokenize the text and assign an integer index to each token, combined with an `Embedding` layer to obtain the vector associated with each token, and then we will use `tf.reduce_mean` to average the representations of the string's tokens.
    - `TextVectorization` will have arguments `max_tokens=None, output_mode='int', vocabulary=<vocabulary created previously>`. We cannot set a limit on the number of tokens because the vocabulary does not coincide with Word2Vec's vocabulary.
    - The `Embedding` layer will have arguments `input_dim=5000, output_dim=300, weights=<created weight matrix>`. 300 is the size of vector representations in this Word2Vec model. The weight matrix will be a bidimensional numpy array with the *i*-th row containing the Word2Vec representation for the *i*-th word in our vocabulary. Notice that the `TextVectorization` will add two extra tokens for out-of-vocabulary tokens and empty tokens. We will set these as two randomly-generated vectors.
    - Computing the average with `tf.reduce_mean` will give us a vector representation for each string.

  - **SentenceBERT.** We explain how we use SentenceBERT in subsection A.2.3 since it was very resource-demanding and we considered more appropiate to compute the sentence representations and save them in a new `Dataset`.

Note. Tweet and user features need an extra detail. `Normalization` **and** `TextVectorization` **layers are unable to output** `RaggedTensors` **yet,** but we found a nice workaround that solves this problem quite easily. `RaggedTensors` are actually a list of values stored in a specified order, and this order can be saved in various formats. For our case, the easiest to handle is the *nested row lengths* format, which in our case simply stores the number of tweets for each piece of news. We wrote a custom layer that takes the flat values of a `RaggedTensor`, applies the layer to the flat values and then reshapes the values into the original shape. This means that `Normalization` will compute the mean and standard deviation from numerical features as if all tweets formed a set instead of being nested inside different news, and `TextVectorization` will vectorize the strings all at once, which actually does not break any rule at all. **Every time one of these layers is used with tweet or user features, we will apply the wrapper on the layer.**

Reducing and concatenating features

Note that while news features have two dimensions (first dimension was created when we batched the `Dataset`, the second one is the feature dimension), tweet and user features have an extra dimension in the middle, corresponding to the number of tweets each news has. Furthermore, this second dimension is not uniform. **We must apply some kind of reduction to summarize the information given by tweets and user profiles.** Our goal is to reduce the tweet dimension to have a bidimensional non-ragged `Tensor` that will be passed to the prediction head. To do that, we follow the steps below:

- **Concatenate news features on the last axis.** With a `Concatenation` layer, we concatenate the news features on the second dimension. This is not mandatory, we could do this right before the prediction head along with the other features, but we no longer need to know where each input comes from. This is not related with the reduction, but we must mention here because we apply it at this point.

- **Concatenate tweet and user features on the last axis.** Again, we can skip this but we no longer need the origin of each column. The last axis here is the third dimension.

- **Apply a reduction layer on the concatenated tweets and user features.** We use `tf.reduce_mean` on the second axis (tweet dimension) although, as explained in 4.2.3, we experimented with a more complex reduction involving an LSTM layer dealing with `RaggedTensors`. **Finally, after many tests, we decided to simplify this stage and just take the average of each feature in the tweet dimension, but we mention this as a remark that more complex reductions can be applied, and this is something unique to this architecture.** We also did some tests taking the maximum with `tf.reduce_max`, and the results were similar to taking the average.

- **Concatenate all the features on the last axis**. After the reduction, we can concatenate the resulting features on the second axis to obtain a non-ragged bidimensional `Tensor` with a fixed shape for all the batches. This `Tensor` will be passed to the prediction head.

Prediction head

We use a *Sequential* model built with a number of interleaving `Dense` and `Dropout` layers. A final `Dense` layer called *prediction* with one output cell and sigmoid activation will output the prediction for each piece of news, which will be interpreted as the probability of the news being *fake* (remember that we are coding fake news as ones and real news as zeros).
After some experimentation, we decided to settle with a prediction model with two `Dense` and `Dropout` layers, excluding the final `Dense` layer, as in figure A.2. The middle `Dense` layers will have arguments

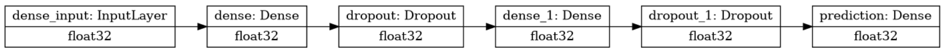


Figure A.2: Tensorflow prediction head.

`n_neurons=48, activation=selu, kernel_regularizer=l2` and the `Dropout` layers will have `dropout_rate=0.2`. These settings will be used throughout all the tests.

Compiling the model

Finally, we compile the models using the *BinaryCrossentropy* loss, since this is a two-class classification problem with binary labels. *Adam* optimizer with `learning_rate=0.001` will be used throughout all the tests (we have tried *SGD* and *adadelta* optimizers, and the results were similar). We will not use any metric other than the model's loss, since we are primarily interested in the test set results and we will use a custom function to log the results.

Tensorflow callbacks

Tensorflow metrics are computed *batchwise* and the average across batches is computed at the end of each epoch. However, we are interested in computing metrics using exactly the same model and using all the samples at once, which can only be achieved using a custom `Callback`. **We wrote a custom `Callback` which we will use during the training phase to compute the model performance in the validation `Dataset` after each epoch.** More precisely, we will compute the accuracy and F1 score using 0.5 as the classification threshold, the AUC for the PR and ROC curves, the optimal threshold for both curves, and the precision, recall, F1 score and accuracy using both optimal thresholds. We define the optimal threshold for the PR curve as the lowest value that maximizes F1 score, and for the ROC curve as the threshold whose point in the ROC curve is closest to $(0, 1)$. **We will also compute all these measures during the evaluation in the test `Dataset`,** taking as optimal thresholds the ones computed in the validation `Dataset`. All these measures will be logged using a `CSVLogger` callback.

Note. We tried to use the `EarlyStopping` callback to stop the training when the validation measures (in particular the F1 score with the optimal PR threshold) stop improving but, unfortunately, did not found any way to watch measures created by another `Callback`. It seems that only metrics are supported. Therefore, **we decided to train the models to a fixed number of epochs,** which will be specified in the testing experiments explained in chapter 5.

To sum up, the complete model has a structure similar to figure A.3. User features are not included in the figure, they would be treated like tweet features and concatenated in the `concatenate_tweets` layer, which would be called `concatenate_tweets_users`.

A.2.3 Modifications to use SentenceBERT

When we started using BERT models to obtain sentence representations, we began by embedding the original BERT model (obtained from Tensorflow Hub) along with the preprocessing steps, replacing the `TextVectorization` layers or the `Embedding` with Word2Vec weight matrix. However, **we soon realized that this approach would not be viable, because each epoch would take more than 20 hours if we were using all the text features from news, tweets and user profiles.** And of course, the hardware requirements were very high, although reasonable using cloud computing. BERT models are quite heavy, and we have more than a million tweets in total. Embedding BERT inside the Functional model is possible if using only news features, but with tweets and user profiles it would take a very long time to train.
We experimented using smaller BERT models. Several BERT models have been published, having either smaller representation sizes (hidden layers with fewer cells) or less transformer blocks. We tried

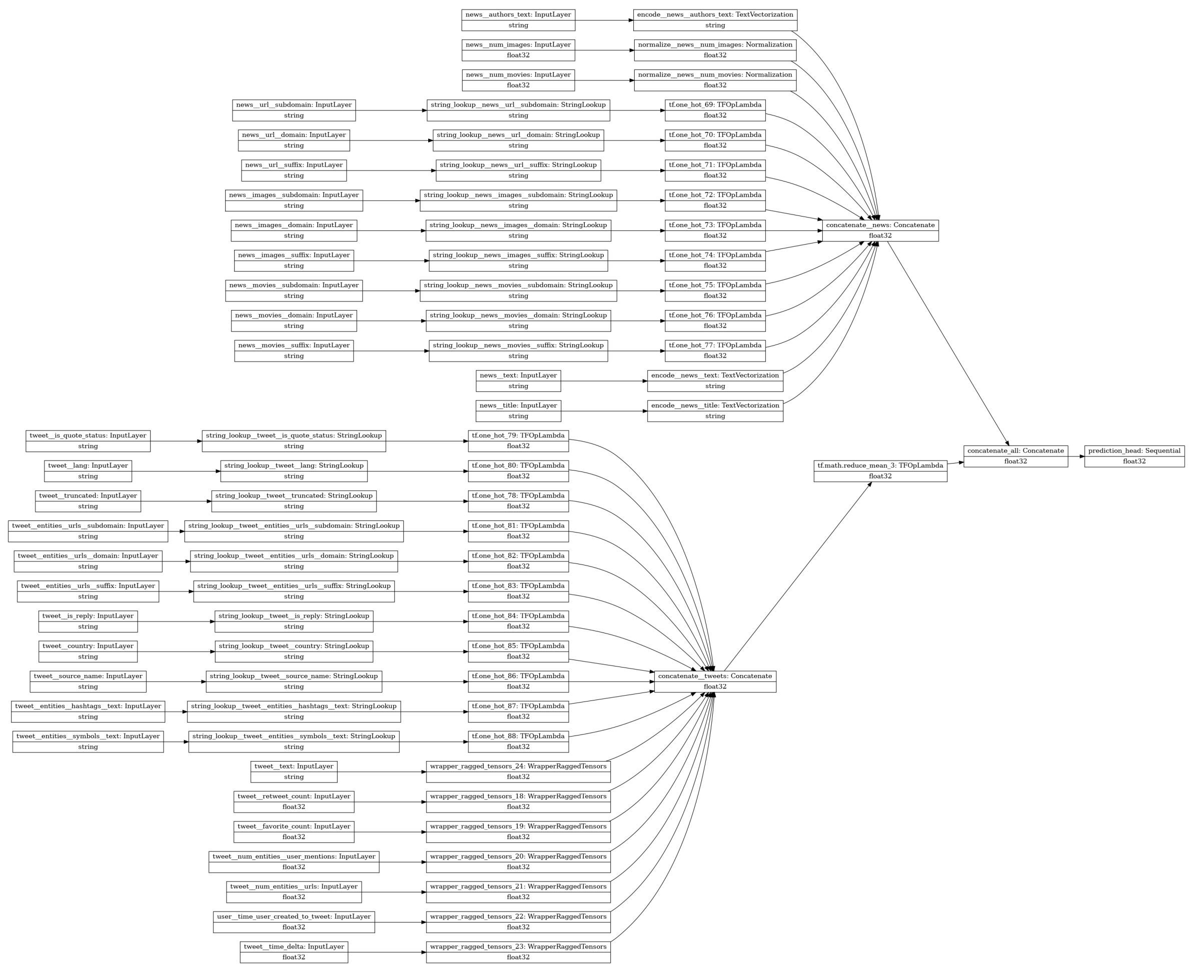

news__authors_text: InputLayer string
encode__news__authors_text: TextVectorization string
news__num_images: InputLayer float32
normalize__news__num_images: Normalization float32
news__num_movies: InputLayer float32
normalize__news__num_movies: Normalization float32
news__url__subdomain: InputLayer string
string_lookup__news__url__subdomain: StringLookup string
tf.one_hot_69: TFOpLambda float32
news__url__domain: InputLayer string
string_lookup__news__url__domain: StringLookup string
tf.one_hot_70: TFOpLambda float32
news__url__suffix: InputLayer string
string_lookup__news__url__suffix: StringLookup string
tf.one_hot_71: TFOpLambda float32
news__images__subdomain: InputLayer string
string_lookup__news__images__subdomain: StringLookup string
tf.one_hot_72: TFOpLambda float32
news__images__domain: InputLayer string
string_lookup__news__images__domain: StringLookup string
tf.one_hot_73: TFOpLambda float32
concatenate__news: Concatenate float32
news__images__suffix: InputLayer string
string_lookup__news__images__suffix: StringLookup string
tf.one_hot_74: TFOpLambda float32
news__movies__subdomain: InputLayer string
string_lookup__news__movies__subdomain: StringLookup string
tf.one_hot_75: TFOpLambda float32
news__movies__domain: InputLayer string
string_lookup__news__movies__domain: StringLookup string
tf.one_hot_76: TFOpLambda float32
news__movies__suffix: InputLayer string
string_lookup__news__movies__suffix: StringLookup string
tf.one_hot_77: TFOpLambda float32
news__text: InputLayer string
encode__news__text: TextVectorization string
news__title: InputLayer string
encode__news__title: TextVectorization string
tweet__is_quote_status: InputLayer string
string_lookup__tweet__is_quote_status: StringLookup string
tf.one_hot_79: TFOpLambda float32
concatenate_all: Concatenate float32
prediction_head: Sequential float32
tweet__lang: InputLayer string
string_lookup__tweet__lang: StringLookup string
tf.one_hot_80: TFOpLambda float32
tf.math.reduce_mean_3: TFOpLambda float32
tweet__truncated: InputLayer string
string_lookup__tweet__truncated: StringLookup string
tf.one_hot_78: TFOpLambda float32
tweet__entities__urls__subdomain: InputLayer string
string_lookup__tweet__entities__urls__subdomain: StringLookup string
tf.one_hot_81: TFOpLambda float32
tweet__entities__urls__domain: InputLayer string
string_lookup__tweet__entities__urls__domain: StringLookup string
tf.one_hot_82: TFOpLambda float32
tweet__entities__urls__suffix: InputLayer string
string_lookup__tweet__entities__urls__suffix: StringLookup string
tf.one_hot_83: TFOpLambda float32
tweet__is_reply: InputLayer string
string_lookup__tweet__is_reply: StringLookup string
tf.one_hot_84: TFOpLambda float32
tweet__country: InputLayer string
string_lookup__tweet__country: StringLookup string
tf.one_hot_85: TFOpLambda float32
tweet__source_name: InputLayer string
string_lookup__tweet__source_name: StringLookup string
tf.one_hot_86: TFOpLambda float32
concatenate__tweets: Concatenate float32
tweet__entities__hashtags__text: InputLayer string
string_lookup__tweet__entities__hashtags__text: StringLookup string
tf.one_hot_87: TFOpLambda float32
tweet__entities__symbols__text: InputLayer string
string_lookup__tweet__entities__symbols__text: StringLookup string
tf.one_hot_88: TFOpLambda float32
tweet__text: InputLayer string
wrapper_ragged_tensors_24: WrapperRaggedTensors float32
tweet__retweet_count: InputLayer float32
wrapper_ragged_tensors_18: WrapperRaggedTensors float32
tweet__favorite_count: InputLayer float32
wrapper_ragged_tensors_19: WrapperRaggedTensors float32
tweet__num_entities__user_mentions: InputLayer float32
wrapper_ragged_tensors_20: WrapperRaggedTensors float32
tweet__num_entities__urls: InputLayer float32
wrapper_ragged_tensors_21: WrapperRaggedTensors float32
user__time_user_created_to_tweet: InputLayer float32
wrapper_ragged_tensors_22: WrapperRaggedTensors float32
tweet__time_delta: InputLayer float32
wrapper_ragged_tensors_23: WrapperRaggedTensors float32

Figure A.3: Tensorflow model with news and tweet features.

several models, and found that the smaller ones were fast enough to train an epoch in 30 minutes, but we were not completely satisfied.

We also had to decide what should we take as the sentence representation, since BERT models output one vector for each token. Several possibilities have been considered (see [RG19], table 6), such as the representation of the `[CLS]` token or the mean across all non-empty tokens. BERT models from Tensorflow Hub contain a *pooled output* that could be used as a sentence representation, although it is not clear how it is computed.

Finally, **we decided to use SentenceBERT-Mean, which is the base BERT model that has been trained on Natural Language Inference data to obtain sentence representations by taking the mean of token embeddings, and try to save a copy of the `Datasets` after vectorizing the text features.** This process would take a long time, but we would only have to do it once, and then we could load the `Dataset` with all the textual features already vectorized, and build the entire model as we have explained in subsection A.2.2 but without any textual vectorization. We believe this is a reasonable approach, since embedding a small BERT model inside our Functional model would need to vectorize the strings on each epoch of each model fit.

Finally, **we needed to use cloud computing, and it took 50 hours to obtain the train, validation and test `Datasets` vectorized and saved to disk using Azure ML on a standard NC6 instance with a Tesla K80 GPU.**

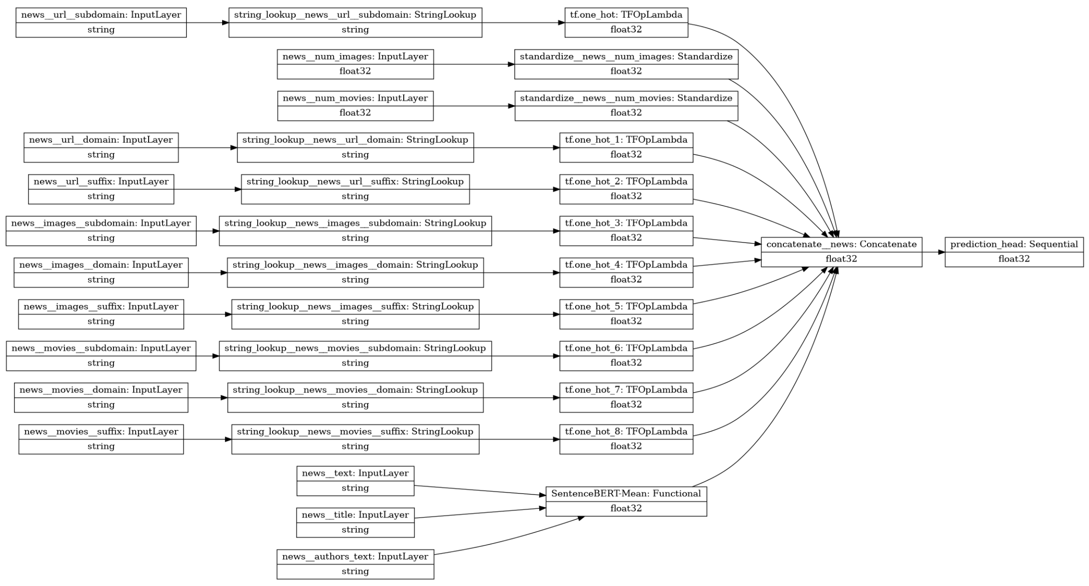

news__url__subdomain: InputLayer
string
string_lookup__news__url__subdomain: StringLookup
string
tf.one_hot: TFOpLambda
float32
news__num_images: InputLayer
float32
standardize__news__num_images: Standardize
float32
news__num_movies: InputLayer
float32
standardize__news__num_movies: Standardize
float32
news__url__domain: InputLayer
string
string_lookup__news__url__domain: StringLookup
string
tf.one_hot_1: TFOpLambda
float32
news__url__suffix: InputLayer
string
string_lookup__news__url__suffix: StringLookup
string
tf.one_hot_2: TFOpLambda
float32
news__images__subdomain: InputLayer
string
string_lookup__news__images__subdomain: StringLookup
string
tf.one_hot_3: TFOpLambda
float32
concatenate__news: Concatenate
float32
prediction_head: Sequential
float32
news__images__domain: InputLayer
string
string_lookup__news__images__domain: StringLookup
string
tf.one_hot_4: TFOpLambda
float32
news__images__suffix: InputLayer
string
string_lookup__news__images__suffix: StringLookup
string
tf.one_hot_5: TFOpLambda
float32
news__movies__subdomain: InputLayer
string
string_lookup__news__movies__subdomain: StringLookup
string
tf.one_hot_6: TFOpLambda
float32
news__movies__domain: InputLayer
string
string_lookup__news__movies__domain: StringLookup
string
tf.one_hot_7: TFOpLambda
float32
news__movies__suffix: InputLayer
string
string_lookup__news__movies__suffix: StringLookup
string
tf.one_hot_8: TFOpLambda
float32
news__text: InputLayer
string
SentenceBERT-Mean: Functional
float32
news__title: InputLayer
string
news__authors_text: InputLayer
string

Figure A.4: Desired Tensorflow model with embedded SentenceBERT. Only news features are shown.

In the next paragraphs we will explain how we used the SentenceBERT model to preprocess the `Datasets` and the modifications needed in the model structure explained in subsection A.2.2 to use these representations as input instead of raw text.

Using SentenceBERT

We mentioned earlier that several BERT models are available in Tensorflow Hub. Unfortunately, SentenceBERT is not one of them, but it is available in HuggingFace's *transformers* model repository (see [Wol+20]). This repository contains many pretrained BERT models for both Pytorch and Tensorflow. For instance, there were other BERT models pretrained to use the `[CLS]` token as the sentence representation, and other variations with different model sizes and tokenizations, models better suited for multilingual texts and so on. Some models were designed for Pytorch, but the *transformers* package contains some tools that allow to use Pytorch models on Tensorflow and viceversa.

We use the model `sentence-transformers/BERT-base-nli-mean-tokens`, which was designed for Pytorch, and we load it using `TFAutoModel.from_pretrained` with parameter `from_pt=True`. This BERT model does not contain the standardization and tokenization steps, but we can use a model available in Tensorflow Hub for their BERT models. The final mean pooling is also not included, and we had to write a custom function to do it. These details are mentioned in the model's information in the repository.

To be exact, we build a Tensorflow Functional model following these steps:

- **Download BERT preprocessor from Tensorflow Hub [1].** This preprocessor is a Tensorflow model that contains the standardization and tokenization for BERT models, which is exactly the same that uses SentenceBERT.
- **Apply the preprocessor** on an `Input` layer with parameters `shape=(), dtype=tf.string`. We will obtain three outputs named `input_ids`, `attention_mask` and `token_type_ids`.
- **Apply SentenceBERT on the preprocessor outputs.** Note that SentenceBERT input names are slightly different to the preprocessor outputs. We can pass the layers as a dictionary.

[1]Current URL: https://tfhub.dev/tensorflow/BERT_en_uncased_preprocess/3

- **Compute the mean of non-empty token representations.** The first output of SentenceBERT contains the token embeddings. We will multiply the embeddings by the `attention_mask` (to skip empty tokens), sum the embeddings and divide by the number of non-empty tokens (or 10^{-9} if there are none).

We perform the processing in Azure ML Studio, on an NC6 instance equipped with a Tesla K80 GPU: for each of the train, validation and test `Datasets`, we batch the `Dataset` (the optimal size is 32), and use `Dataset.map` to preprocess each batch. The output for each batch (dictionary storing `Tensors` or `RaggedTensors`) will be another dictionary with the same keys, and the values will be either the same (for non-textual features) or the output of the Functional model we just created (for textual features). Then, we unbatch the `Dataset` with `Dataset.unbatch` and we save the `Dataset` using `tf.data.experimental.save` and copy the folder to the *output* folder.

To use the vectorized `Datasets`, we need to load them with `tf.data.experimental.load`, using a slightly modified version of the output signature we used when we created the original `Datasets` from a Python generator in A.2.1. To be precise, we have to change the `TensorSpec` objects by `RaggedTensorSpecs` for each textual feature, with parameters `shape=(768,), dtype=tf.float32`.

Also, in the classification model, we have to **remove the preprocessing steps for textual features, and modify the `Input` layer of these features,** setting the parameters `shape=(768,)`, `dtype=tf.float32`, and setting `ragged=True` for tweets and user profiles' textual features.

Note. 768 is the size of BERT base embeddings.

Appendix B

Result Tables

This appendix contains the results of the experiments that were carried out in chapter 5. The results in terms of F1 scores were already shown in section 5.2, we include here the results obtained in the test set, in terms of accuracy and, for the Deep Learning architecture, also the precision-recall AUC and ROC AUC. All the scores are multiplied by 100, as in section 5.2. We follow a structure similar to section 5.2.

Most of the discussion in section 5.2 is still valid when looking at the accuracy or the AUCs. Some remarks do not hold, since a model might have high accuracy and low F1 score if precision is low. When needed, we will make the necessary remarks.

B.1 Non Deep Learning

We leave here the average test accuracy scores obtained when using all news, first, and when using only news with tweets, later. Recall that, in tables B.2 and B.4, textual features are encoded using the bag-of-words Vector Space Model, since we saw in subsection 5.2.2 that it performs better and more consistently than the other representation techniques.

B.1.1 All news

All the remarks made in 5.2.2 are still valid. However, table B.2 shows more clearly that numerical features are not very useful.

	Algorithm	LR	LIN-SVM	RBF-SVM	RF	LGBM	XGB
Source	Vectorization						
GossipCop	Bag of Words	85.0	83.6	86.6	85.5	**89.0**	88.7
	Frequency	83.7	81.3	86.3	86.0	88.0	88.1
	TF-IDF	83.2	81.8	86.5	85.7	87.9	88.0
PolitiFact	Bag of Words	90.1	81.7	92.6	89.1	95.1	**96.3**
	Frequency	90.1	84.4	82.7	88.4	95.1	92.6
	TF-IDF	87.7	83.5	74.1	90.1	92.6	90.1

Table B.1: Average test accuracy by text vectorization and algorithm (all news, non DL vs DL).

Source	Algorithm Feature type	LR	LIN-SVM	RBF-SVM	RF	LGBM	XGB
GossipCop	Numerical	75.8	75.8	75.8	77.7	78.7	78.8
	Categorical	82.7	79.7	82.9	82.7	82.8	82.7
	Textual	82.9	81.3	86.3	84.6	87.3	86.9
	All	85.0	83.6	86.6	85.5	**89.0**	88.7
PolitiFact	Numerical	67.9	67.9	69.1	82.7	81.5	81.5
	Categorical	74.1	70.9	70.4	73.1	64.2	69.1
	Textual	90.1	88.1	88.9	91.4	93.8	91.4
	All	90.1	81.7	92.6	89.1	95.1	**96.3**

Table B.2: Average test accuracy by feature type and algorithm (all news, non DL vs DL).

B.1.2 News with tweets

The remarks about the comparison by text vectorization and algorithm are still valid. However, in table B.4, we notice that, in the PolitiFact collection, when using numerical or categorical features, the accuracy increases with respect to using all news (table B.2). Also, logistic regression and both SVM obtain a mean accuracy higher than the dummy baseline, which means that low precision is penalizing their F1 scores.

Source	Algorithm Vectorization	LR	LIN-SVM	RBF-SVM	RF	LGBM	XGB
GossipCop	Bag of Words	84.8	82.1	85.6	85.0	**87.9**	87.1
	Frequency	83.3	81.1	85.4	85.2	87.7	86.7
	TF-IDF	83.1	80.8	84.9	85.2	87.6	87.1
PolitiFact	Bag of Words	88.9	92.1	85.7	86.3	90.5	90.5
	Frequency	87.3	88.9	69.8	86.7	**93.7**	92.1
	TF-IDF	87.3	91.7	73.0	87.9	**93.7**	**93.7**

Table B.3: Average test accuracy by text vectorization and algorithm (non DL vs DL).

Source	Algorithm Feature type	LR	LIN-SVM	RBF-SVM	RF	LGBM	XGB
GossipCop	Numerical	74.8	74.8	74.8	76.7	78.8	78.8
	Categorical	81.8	79.7	82.1	82.4	81.9	81.9
	Textual	81.6	79.4	85.1	83.6	86.5	85.9
	All	84.8	82.1	85.6	85.0	**87.9**	87.1
PolitiFact	Numerical	68.3	68.3	69.8	81.9	85.7	84.1
	Categorical	77.8	79.4	73.0	76.8	73.0	74.6
	Textual	87.3	85.4	81.0	87.6	**92.1**	87.3
	All	88.9	**92.1**	85.7	86.3	90.5	90.5

Table B.4: Average test accuracy by feature type and algorithm (non DL vs DL).

B.2 Deep Learning

We leave here the average test accuracy scores, precision-recall AUCs and ROC AUCs obtained with our Deep Learning architecture. All the remarks made in 5.2.3 are still valid here. Recall that, in tables B.8, B.9 and B.10, textual features are encoded using the bag-of-words model, as we saw in 5.2.3 that it obtains good results and is consistent across most combinations.

Source	Feature origin / Vectorization	News	Tweets	Users	News+Tweets	News+Users	All
GossipCop	Bag of Words	82.7	**96.0**	95.7	92.5	93.7	94.8
	Frequency	**84.3**	**96.0**	**96.1**	90.2	91.7	92.9
	TF-IDF	84.0	92.2	**96.1**	87.8	91.1	92.0
	Word2Vec	78.2	95.5	94.5	**94.3**	**94.4**	**95.4**
	SentenceBERT	77.9	93.8	93.8	90.8	93.3	93.7
PolitiFact	Bag of Words	**93.0**	93.0	**85.7**	**95.6**	**94.9**	**95.2**
	Frequency	90.8	89.5	81.3	91.1	88.9	88.9
	TF-IDF	87.9	90.2	83.5	92.1	90.8	91.7
	Word2Vec	89.5	87.0	74.0	95.2	87.0	92.7
	SentenceBERT	83.8	**94.9**	71.1	88.6	84.4	87.0

Table B.5: Average test accuracy by text vectorization and feature origin (DL).

Source	Feature origin / Vectorization	News	Tweets	Users	News+Tweets	News+Users	All
GossipCop	Bag of Words	**76.2**	**98.0**	95.9	92.9	94.1	96.0
	Frequency	76.1	97.9	**96.3**	89.0	88.8	91.3
	TF-IDF	76.1	91.9	95.6	84.2	89.5	91.1
	Word2Vec	61.7	97.1	93.1	**95.9**	93.1	**96.7**
	SentenceBERT	71.5	94.4	93.0	91.6	**94.3**	95.0
PolitiFact	Bag of Words	**99.4**	**98.6**	**96.0**	**99.6**	**99.5**	**99.6**
	Frequency	91.4	98.3	93.4	91.7	92.5	88.8
	TF-IDF	92.5	97.3	85.4	93.1	89.4	92.6
	Word2Vec	89.7	92.5	81.8	96.6	90.9	96.4
	SentenceBERT	96.4	97.7	74.8	97.2	96.0	97.2

Table B.6: Average test PR AUC by text vectorization and feature origin (DL).

Source	Feature origin Vectorization	News	Tweets	Users	News+Tweets	News+Users	All
GossipCop	Bag of Words	86.4	**99.2**	98.4	96.4	97.5	98.2
	Frequency	85.5	**99.2**	**98.6**	93.9	94.3	95.7
	TF-IDF	**86.7**	96.7	98.5	91.4	95.2	96.0
	Word2Vec	76.6	98.9	97.6	**98.7**	**98.0**	**99.0**
	SentenceBERT	83.3	97.2	97.5	96.0	97.7	97.9
PolitiFact	Bag of Words	**99.3**	**98.4**	**94.7**	**99.6**	**99.4**	**99.5**
	Frequency	94.2	98.0	93.1	94.1	94.2	93.1
	TF-IDF	93.4	97.4	87.2	94.9	92.6	94.6
	Word2Vec	92.6	93.7	81.4	97.0	93.5	96.6
	SentenceBERT	95.3	97.9	78.1	96.4	95.0	96.6

Table B.7: Average test ROC AUC by text vectorization and feature origin (DL).

Source	Feature origin Feature type	News	Tweets	Users	News+Tweets	News+Users	All
GossipCop	Numerical	29.0	80.1	78.7	79.9	79.0	84.9
	Categorical	79.6	95.4	95.4	**95.0**	**94.6**	**96.4**
	Textual	**83.3**	89.4	**96.5**	86.1	92.1	92.6
	All	82.7	**96.0**	95.7	92.5	93.7	94.8
PolitiFact	Numerical	54.0	79.4	61.3	79.0	60.3	81.9
	Categorical	74.6	84.4	76.5	90.5	86.3	90.8
	Textual	**93.3**	82.2	77.5	92.7	**94.9**	90.8
	All	93.0	**93.0**	**85.7**	**95.6**	**94.9**	**95.2**

Table B.8: Average test accuracy by feature origin and feature type (DL).

Source	Feature origin Feature type	News	Tweets	Users	News+Tweets	News+Users	All
GossipCop	Numerical	29.7	67.4	64.1	67.8	64.0	72.5
	Categorical	61.4	96.6	94.2	**96.4**	**95.4**	**97.4**
	Textual	75.0	87.1	95.7	81.3	91.7	92.7
	All	**76.2**	**98.0**	**95.9**	92.9	94.1	96.0
PolitiFact	Numerical	63.4	90.5	63.3	86.6	63.7	83.4
	Categorical	89.2	94.0	83.8	96.4	94.1	98.5
	Textual	98.6	91.9	87.7	98.2	98.9	98.3
	All	**99.4**	**98.6**	**96.0**	**99.6**	**99.5**	**99.6**

Table B.9: Average test PR AUC by feature origin and feature type (DL).

Source	Feature origin Feature type	News	Tweets	Users	News+Tweets	News+Users	All
GossipCop	Numerical	55.4	82.8	82.0	83.1	82.1	86.0
	Categorical	76.6	98.7	97.9	**98.9**	**98.4**	**99.1**
	Textual	85.4	93.7	**98.5**	89.5	96.5	96.7
	All	**86.4**	**99.2**	98.4	96.4	97.5	98.2
PolitiFact	Numerical	63.7	91.0	66.5	90.1	66.1	87.2
	Categorical	86.2	93.9	86.5	96.6	94.7	98.3
	Textual	98.5	90.5	85.8	97.8	98.6	98.0
	All	**99.3**	**98.4**	**94.7**	**99.6**	**99.4**	**99.5**

Table B.10: Average test ROC AUC by feature origin and feature type (DL).