ELEC-Y-591 Machine Learning and Big Data Processing

June 2019

Rumour Detection on Social Media

Authors :

Detemmerman	Charles
Joukovsky	Boris
Rohn	Alexander
Storrer	Laurent

Contents

Individual contributions

- State of the art: Alexander Rohn
- Dataset gathering: Charles Detemmerman and Boris Joukovsky
- Events representation: Laurent Storrer and Alexander Rohn
- Features extraction: Laurent Storrer and Alexander Rohn
- Neural networks and training: Charles Detemmerman and Boris Joukovsky

1 Introduction

With the emergence of social media platforms, people all around the world have drastically changed their approach towards news and information. On one hand, social media platforms can be seen as a tool to connect people, and promote freedom of speech as well as democracy. Social media platforms were used during the Tunisian Arab Spring, which led to the ousting of the Ben Ali family, and the restoring of democracy [1]. On the other hand, social media platforms can be seen as a tool to create divisions, and manipulate people. During the U.S. election of 2016, which led to Donald Trump becoming U.S. President, Russian hackers and trolls used social media and so-called "fake news" (i.e. rumours) to influence American voters [2]. Any social media platform user can read and share information on the platform, anywhere at any time. Consequently, on these platforms, rumours can spread extremely fast. It is thus crucially important to develop methods for detecting rumours on social media.

2 State of the Art

Rumour detection in social media can be handled in a lot of different ways. In this section, different approaches towards rumour detection will be outlined.

R. Moin et al. [3] use a framework based on inquiry comment detection. In their paper, an inquiry comment is defined as a sentence with specific characteristics, such as:

- It ends with a question mark "?".
- It starts with an intransitive verb (e.g. "was", "were" or "would").
- It contains the words "real" or "really", it contains the word "true" or "unconfirmed".

It should be noted that the characteristics listed hereabove correspond to a non-exhaustive list of the characteristics chosen by R. Moin et al. to define an inquiry comment. The focus of R. Moin et al. lies on predicting if a given Facebook post corresponds to a rumour, or not. In order to do so, their approach is quite simple. To begin with, all the comments related to the given post are gathered, and ordered into 2 distinct categories: Inquiry and Non-Inquiry comments. If the ratio between both category sizes is above a pre-defined threshold, the post is considered as a rumour. The method was tested on a set of labelled Facebook posts, and an accuracy of 70% was obtained.

J. Ma et al. [4] use an approach based on a Recursive Neural Network (RNN). In their approach, each social media post is represented by a so-called "event". An event corresponds to the post itself and all its re-posts. The goal is to use a RNN to classify the event as rumour or non-rumour. In order to do so, each event is represented by a time-series, and separated into fixed-length non-empty time intervals. Each time interval contains a given number of re-posts. The re-posts in each time-interval are represented by their *tf-idf* (Text Frequency-Inverse Document Frequency) score. The time-series, which is composed of several feature vectors containing *tf-idf* scores, is then fed to a RNN. The output of the RNN will determine if the event is a rumour or not. J. Ma et al. trained and tested different types of RNNs on a Twitter and Weibo dataset, and obtained accuracies above 80%.

In the paper of N. Ruchansky et al. [5] the same type of RNN-based approach is used. A social media post is also represented by an event, and then decomposed into time intervals. N. Ruchansky et al. suggest to use *doc2vec* instead of *tf-idf* to represent the re-posts in each time interval. They also suggest to add user-related information to the feature vectors, as well as information such as the total number of re-posts. The feature vectors are then fed to a RNN. This method was tested by N. Ruchansky et al. on a Twitter and Weibo dataset, and reached accuracies of 89% and 95% respectively.

3 Methods

3.1 Events representation

In terms of how to represent the input data, the method of J. Ma et al. [4] was considered. The datasets used by J. Ma et al are publicly available [1], and were used in the course of this project in order to better compare the results with the existing work. Of the two available datasets, Twitter and Weibo, Twitter was selected in order to be able to interpret the input data, as none of the authors of this work speak mandarin. The Twitter dataset contains so-called *events*. An event is composed of an original post (i.e. tweet) and all its re-posts (i.e. re-tweets). The objective is to classify events as rumour or non-rumour. In order to perform classification, each event has to be represented by a set of features. The feature extraction techniques are discussed in section 3.2. Prior to feature extraction, the tweets in each event have to be cleaned. The details on how the Twitter dataset was gathered and on how the cleaning process occurs can be found in Appendix C. Once the feature extractor is trained, the extraction on each event can be performed in different ways.

- **One block for ANN:** The feature extraction is performed on each event separately, producing one vector per event. Each feature vector can then be fed to an classical Artifical Neural Network (ANN).
- **Cutting in intervals for RNN:** the event can also be cut in time intervals, i.e. transformed into a time-series. The intervals all have the same length. However, this length can be calculated in different ways. The starting point of this calculation is the event timeline, i.e. the time interval between the first and last post's posting time.
  - **Fixed intervals:** the timeline is cut in N intervals. Each interval can contain a certain number of posts, or it can be empty. Feature extraction is then performed on each interval, including the empty ones, giving one feature vector per interval. Those vectors are then fed to a Recurrent Neural Network (RNN).
  - **Variable intervals:** the timeline is initially cut in N intervals, each spanning a time l. Then the algorithm halves l, potentially several times, to look for the series of non-empty intervals covering the longest time span possible. The details of the update of l are summarized in Figure 7 in Appendix A. This series of intervals is saved, and the other intervals (containing other re-posts) are discarded. Feature extraction is then applied on the intervals of the saved series only, giving one vector per interval. Again, those vectors are then fed to a RNN. This is the approach followed by J. Ma et al. [4].

[1]http://alt.qcri.org/ wgao/data/rumdect.zip

In both of the cases that use the time-series representation, N was set to 12. Features from these groups of tweets can now be extracted using one of the following methods described in section 3.2.

3.2 Feature extraction

The feature extractors presented in this section are trained on the whole set of events belonging to the training set only. The concatenated strings of all the tweets corresponding to a single event is denoted as a *document*. Consequently, the feature extractors are trained on a set of such documents.

3.2.1 TF-IDF

TF-IDF is the accronym of Term Frequency - Inverse Document Frequency. It is a metric allowing to assess the importance of a word in a document that belongs to a collection of documents [6]: the higher its *tf-idf* score, the higher its importance. In other words, each word w_i has a *tf-idf* score per document j, which is the product of a *tf* score and an *idf* score:

- The *tf* (term frequency) score is the frequency of occurrence of the word in a document. Each word has thus one tf score per document. The tf score a word w_i in a document j is [6]:

$$tf_j(w_i) = \frac{n_j(w_i)}{\sum_{i'} n_j(w_{i'})} \tag{1}$$

  where $n_j(w_i)$ is the number of occurence of word w_i in document j. Equation 1 shows that the more a word appears in a given document, the higher his tf score is in this document.

- The *idf* (inverse document frequency) score quantifies how much a given word appears accross all documents in the collection. There is thus one idf score per word accross the whole collection. If there are D documents, it is expressed as [6]:

$$idf(w_i) = log\left(\frac{D}{d(w_i)}\right) \tag{2}$$

  where $d(w_i)$ is the number of documents containing word w_i. A rare word in the collection, *i.e.* a word that only appears in a few documents, carries more information and so by Equation 2 it has a higher idf score [6].

Combining those two scores gives the *tf-idf* score of the word w_i in a document j [6]:

$$\text{tfidf}_j(w_i) = tf_j(w_i)\ idf(w_i) \tag{3}$$

This *tf-idf* metric makes a balance between the number of occurences of the word in the document and its rarity in the collection of documents. In this work, a pre-implemented *tf-idf* feature extractor, TfidfVectorizer [7] is used, from the `scikit-learn` [8] toolkit. The idf score of each word is computed once accross the whole collection of documents/events, with the function `fit` of TfidfVectorizer. The K words with the highest idf score are kept as the *vocabulary*. The value of K is discussed in section 4.1, but is set to 2500 as a default value. Then for each event, depending on the event representation:

- In the ANN approach: for each event (document) the tf scores of event words present in the vocabulary are computed, and then multiplied with the previously computed idf scores, giving one vector of size $K \times 1$ with 0s for vocabulary words not present in the current event, and positive *tf-idf* scores otherwise. This is done with the function `transform` of TfidfVectorizer [7]. This vector of *tf-idf* scores corresponds to the feature vector of the event. Since there are not a lot of words in one document compared to the K words in the vocabulary, this output vector is very sparse.

- In the RNN approach: the tf score is computed (for the words in the vocabulary) for each interval separately, and then multiplied with the previously computed idf score, again with the function `transform`. There is thus one feature vector, again very sparse, for each interval.

3.2.2 Word2Vec

Word2vec is a word embedding technique allowing to predict words based on their context. It is more advanced than *tf-idf* because it captures the meaning of each word. Each word is represented by a vector of features, that are learned based on the words in the document [9]. *Word2vec* is not used in this work because it adds one dimension to the features compared to *tf-idf*: in *tf-idf* each word was represented by a scalar (the *tf-idf* score), while in *word2vec* each word is represented by a vector. Using *word2vec* would thus increase drastically the amount of parameters to train in the neural network. *Word2vec* was thus not used in this work, but it is still explained in Appendix B in order to facilitate the understanding of the *doc2vec* technique.

3.2.3 Doc2vec

Doc2vec [10] is used to produce a vector representation of a document. Thus, in the ANN approach of this work, it would produce one vector per event/document. *Doc2vec* follows the same principle as word2vec (cf. Appendix B). The CBOW architecture is considered here: the same one-hot encoded words are fed as input, but one extra input is also added: the one-hot encoded document index. For example if there are D documents, the d-th document is represented by a vector with 0s everywhere except at index d where there is a 1. This vector is thus added as extra input. It has a specific weight matrix W_D of size $D \times N$ to map it to the hidden layer [11]. The network is trained similarly as in word2vec, and the d-th row of this matrix W_D is the vector of size $1 \times N$ representing the document d [11], which can then be fed to the ANN. The value of N (number of neurons in the hidden layer) is thus similar to the K of *tf-idf*, because it is also the size of the vector representing the document. A pre-implemented *doc2vec* feature extractor from the `gensim` librairy is used [12]. The used parameters are summarized in the table herebelow [13].

	Voc. size V	Feat. vector size N (or K)	Learning rate	Epochs
Doc2vec	All words	2500	0.025	100

Table 1: *Doc2vec* parameters

3.3 Studied cases

After this explanation of the events structure and the feature extraction methods, the different cases to investigate are summarized herebelow:

Case 1	**Case 2**	**Case 3**	**Case 4**
ANN & *tf-idf*	ANN & *doc2vec*	RNN & fixed intervals & *tf-idf*	RNN & variable intervals and *tf-idf*

Table 2: Studied cases

3.4 Network architectures

3.4.1 ANN

As mentioned previously, two main classes of neural network architectures were used in this work, a standard Artificial Neural Network (ANN), and a Recurrent Neural Network (RNN), as used in J. Ma et al. [4]. The ANN architecture is a simple multilayer perceptron model, as described below:

- **Input layer:** Vector of size K containing one of the feature types described previously
- **Embedding layer:** Fully-connected hidden layer with ReLU activation, L2 regularization and dropout regularization. The embedding layer is necessary to perform a dimensionnality reduction step and avoid overfitting.
- **Hidden layers:** Additional fully-connected hidden layers using the same hyperparameters as the embedding layer.
- **Output layer:** Softmax-activated fully-connected classification layer (2 classes) with L2 regularization.

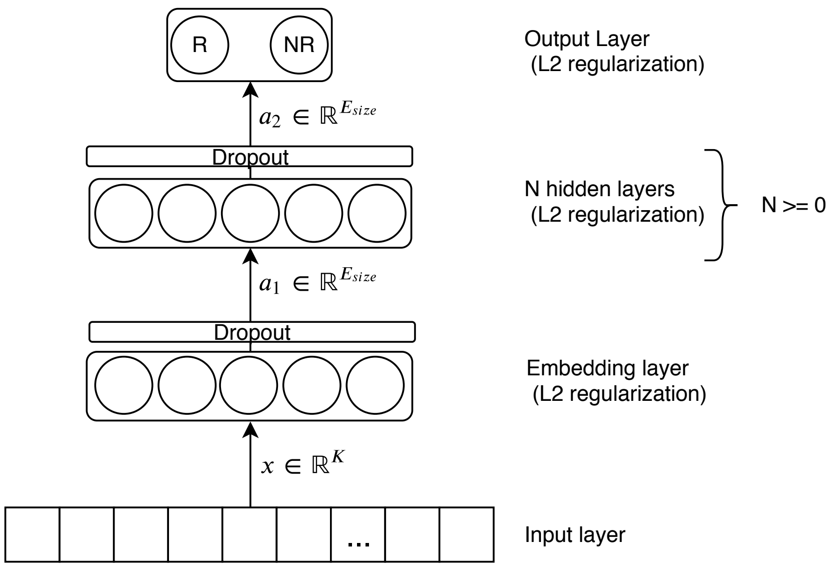
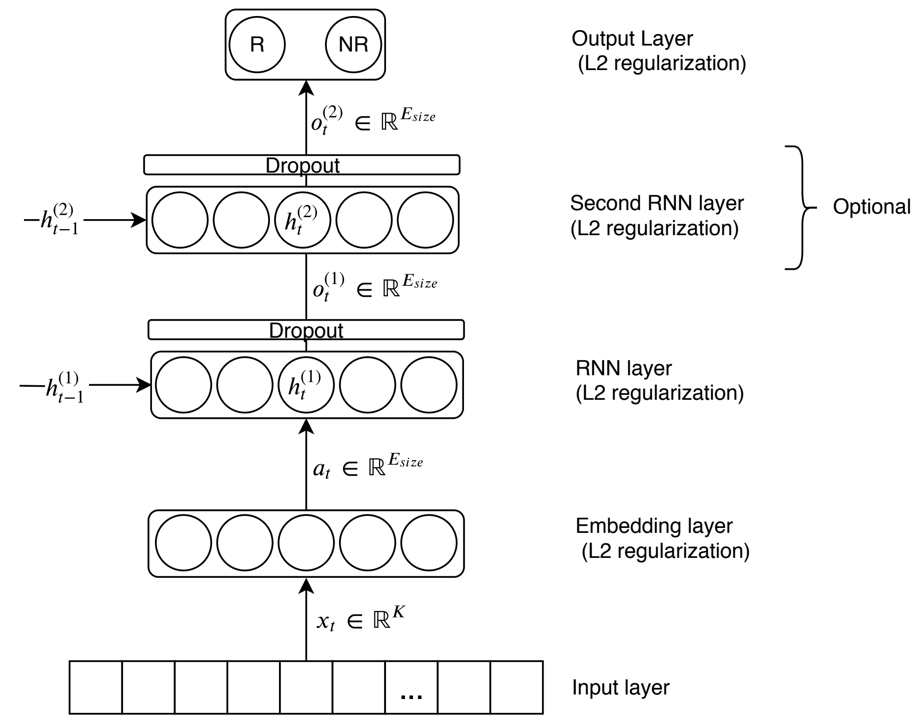


Figure 1: ANN example architecture with 2 hidden layers

An example configuration is shown on Figure 1, where one can identify the following hyperparameters which were compared to find the optimal network configuration: feature vector size (K), embedding layer size (E_{size}), number of hidden layers (N), dropout rate and regularization coefficient. Note that the dropout was shown as a network layer, although it is not used during evaluation.



R
NR
Output Layer (L2 regularization)
$o_t^{(2)} \in \mathbb{R}^{E_{size}}$
Dropout
$-h_{t-1}^{(2)}$
$h_t^{(2)}$
Second RNN layer (L2 regularization)
Optional
$o_t^{(1)} \in \mathbb{R}^{E_{size}}$
Dropout
$-h_{t-1}^{(1)}$
$h_t^{(1)}$
RNN layer (L2 regularization)
$a_t \in \mathbb{R}^{E_{size}}$
Embedding layer (L2 regularization)
$x_t \in \mathbb{R}^K$
...
Input layer

Figure 2: Generic RNN architecture for any of the RNN models (basic, GRU, LSTM)

,

3.4.2 RNN

The second network type used is the RNN. A RNN is a class of stateful artificial neural network, which works on sequences of data (e.g. time sequences) by updating an internal state at each step. The basic operation of an RNN can be described as follows:

$$\begin{aligned} h_t &= \sigma(Ux_t + Xh_{t-1} + b) \\ o_t &= Vh_t + c \end{aligned} \tag{4}$$

Where, at each time step, the model uses the input sequence $(x1, ..., x_T)$ to update the internal hidden states $(h_1, ..., h_T)$, and generates the output vector $(o_1, ..., o_T)$ from the hidden states. U, W, V are the weight matrices, b and c the bias vectors, and σ the non-linearity function. The hyperbolic tangent $tanh(.)$ is the non-linearity function used in [4]. However, in this work, a ReLU activation is used. This basic RNN was deployed with an embedding layer, and dropout regularization as shown in the generic RNN architecture of Figure 2.

Two other RNN models were used for comparison, the Long-Short Term Memory (LSTM) and Gated Recurrent Unit (GRU). These are a bit more complex, with additional gates controlling the rate at which memory is forgotten and at which a new input will affect the memory. The LSTM model is reproduced in in Equation 5, and the GRU model, which is slightly simpler, is reproduced in Equation 6. Note that in the case of the GRU, the output is h_t. The basic RNN was used only with a single layer configuration, whereas a two RNN layers configuration was also tried for the GRU and LSTM, as was done in [4].

LSTM

$$\begin{aligned}
i_t &= \sigma(x_t W_i + h_{t-1} U_i + c_{t-1} V_i) \\
f_t &= \sigma(x_t W_f + h_{t-1} U_f + c_{t-1} V_f) \\
\tilde{c_t} &= ReLU(x_t W_c + h_{t-1} U_c) \\
c_t &= f_t c_{t-1} + i_t \tilde{c_t} \\
o_t &= \sigma(x_t W_o + h_{t-1} U_o + c_{t-1} V_o) \\
h_t &= o_t \cdot ReLU(c_t)
\end{aligned} \tag{5}$$

GRU

$$\begin{aligned}
z_t &= \sigma(x_t U_z + h_{t-1} W_z) \\
r_t &= \sigma(x_t U_r + h_{t-1} W_r) \\
\tilde{h_t} &= ReLU(x_t U_h + (h_{t-1} \cdot r_t) W_h) \\
h_t &= (1 - z_t) \cdot h_{t-1} + z_t \tilde{h_t}
\end{aligned} \tag{6}$$

3.5 Training details

3.5.1 Data splitting and k-folds cross-validation

The gathered *Twitter* dataset contains a total of **992 events** with an equal repartition between rumours and non-rumours. These events cumulate a total of **651,255 tweets**. The dataset is split into a training set (85%) and a test set (15%), using a predetermined division of the data, ensuring that test set remains unchanged and allowing to fairly compare the different architectures.

Because the dataset is not very large in terms of events, it is chosen to train the networks using the **k-folds cross-validation** method, allowing to exploit a maximum amount of data and to better estimate the classification performance with respect to small variations of the training set. This method consists in training k versions of the network on alternating $\frac{k-1}{k}$ fractions of the original training set, while the remaining $\frac{1}{k}$ fraction constitutes the validation set. In this case, $k = 5$ was chosen as a trade-off between the amount of networks to train and a sufficient size of the training sets.

As for the final evaluation on the test set, a custom ensemble method is used: the 5 different network versions resulting from the k-folds cross-validation are exploited to predict 5 separate outputs. The binary label y is assigned to the sample $\mathbf{x}$ according to the following heuristic rule: the probability of $\mathbf{x}$ being a rumor is given by the average of the class probability outputs of the 5 trained networks, as defined in equation 7, where θ_k represents the parameters the different network versions.

$$p(y|\mathbf{x}) = \sum_{k=1}^{5} p(y|\mathbf{x}, \theta_k) p(\theta_k) = \frac{1}{5} \sum_{k=1}^{5} p(y|\mathbf{x}, \theta_k) \tag{7}$$

3.5.2 Losses and metrics

As explained earlier, the binary labels are represented as one-hot encoded vectors. For example, the label 0 corresponding to a non-rumour is encoded as $[1\ 0]^T$, while a rumour is encoded as

$[0\ 1]^T$. Due to this representation choice, the `categorical_crossentropy` function of *Keras* must be used, instead of the classical `binary_crossentropy`. For a dataset of size N with predictions $\mathbf{y}$ of dimension $(N, 2)$ and with ground-truth labels $\hat{\mathbf{y}}$, this loss function can be written as in equation 8. Additionally, the regularization term is added which acts on the parameter vector $\boldsymbol{\theta}$. Note that the following loss function is computed on the output softmax class-probabilities:

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}, \boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=0}^{1} y_j^{(i)} \log(\hat{y}_j^{(i)}) + \lambda \|\boldsymbol{\theta}\|_2^2 \tag{8}$$

Additionally, for evaluating the network, the *accuracy* metric is used and simply represents the ratio of correctly classified samples over the dataset size. In the following equation 9, $\delta(i, j)$ corresponds to the Kronecker delta:

$$Acc(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{N} \sum_{i=1}^{N} \delta(\text{argmax}(\hat{\mathbf{y}}^{(i)}), \text{argmax}(\mathbf{y}^{(i)})) \tag{9}$$

3.5.3 Optimization and monitoring

Training the networks is done using the *RMSProp*[14] optimizer with a learning rate of 10^{-4}, which is basically a stochastic gradient descend algorithm with momentum. The batch size is set to 32. The `Checkpoint` callback of *Keras* is used to retain the best network based on the maximum *validation categorical accuracy* score. Nevertheless, to avoid overfitting, `EarlyStopping` is used to interrupt the training process when the *validation loss* rises after having reached a minimum. Finally, `TensorBoard` is also used to monitor the training process online.

4 Results

Before selecting the best performing network configuration for each of the four cases mentioned in section 3.3, intermediate results will be given trough a comparative study of the network hyperparameters such as the number of hidden layers and units (embedding size), the regularization parameter, etc. In the following experiments, networks are compared based on the mean and standard deviation of the obtained accuracies for the 5 folds, while the test accuracy is computed based on the combination of the 5 network outputs as expressed in equation 7. A summary of the final results will be given in section 4.4.

4.1 Case 1: ANN with tf-idf

The features extracted using *tf-idf* with $K = 2500$ are used to train 1-hidden layer and 2-hidden layers networks (here the embedding layer is included in the hidden layers count). The embedding size is varied in order to estimate the optimal network configuration. Results are reported below in figure 3. The regularization λ is set to 10^{-1}, the dropout rate to 0.5 and the learning rate to 10^{-1}.

In both cases, overfitting can be observed when the number of hidden units is increased: the training curve is rising while the test accuracy stays constant. By looking at the validation

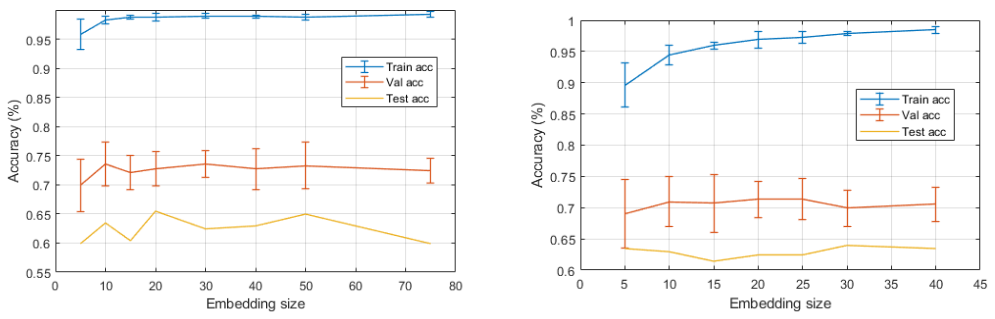


Figure 3: Embedding size study. Left: 1 hidden layer ANN. Right: 2 hidden layers ANN

curve, the 1-layer configuration seems to be superior to the other case. However, the gap between the validation and the test accuracy is also larger, indicating that this configuration lacks in generalizability.

The effect of the number of tf-idf features is also studied through the following experiments: 2 ANN with 2 hidden layers are trained using an embedding size of 20, and features are extracted (1) with $K = 1000$ and (2) with $K = 2500$. Results are indicated below in table 3 and shows that using a higher feature size increases the performance while reducing the network variability.

K	Val acc mean (%)	Val acc std	Test acc (%)
1000	66.5	5.2	59.4
2500	70.2	3.4	64.0

Table 3: Studying the number of features extracted using *tf-idf*

The optimization of the dropout rate and the regularization parameter λ is not presented here as the following section 4.2 already gives an example of such a study. Nevertheless, tweaking these parameters led to the choice of $\lambda = 10^{-1}$ and drop rate $= 0.5$.

4.2 Case 2: ANN with doc2vec

In this case, the features are extracted with *doc2vec* using $K = 2500$. Similarly to the previous case, various configurations of the network are tested using multiple embedding sizes, for 1 and 2 hidden layers. Results can be found below in figure 4. The learning rate is set to 10^{-4}, the dropout rate to 0.5 and the regularization parameter λ to 10^{-1}.

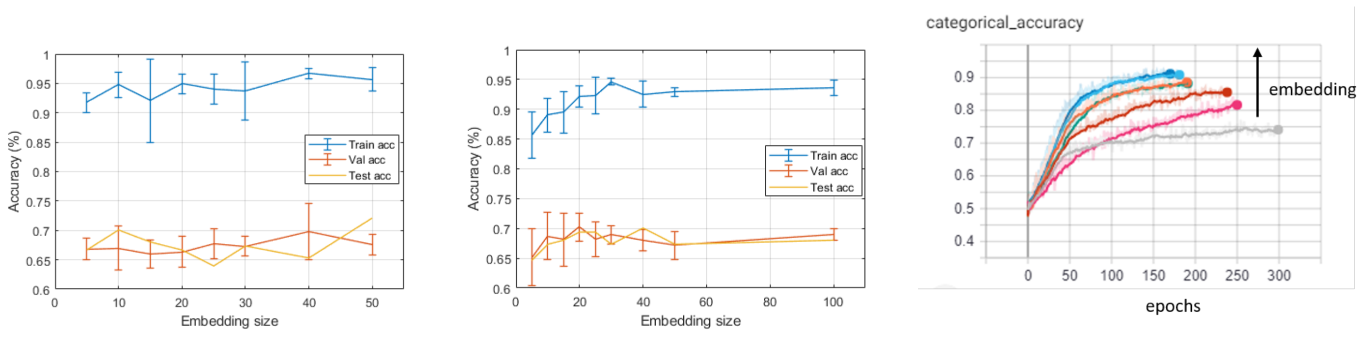


Figure 4: Embedding size study. Left: 1-layer ANN. Middle and Right: 2-layers ANN

At a first glance, it seems that the 2-layer ANN is able to reach slightly higher scores than its 1-layer counterpart. This happens for an embedding size which seems optimal around 20 in the 2-layer case. It also appears that increasing the embedding size may increase overfitting, as the plots indicate that the training accuracy becomes high while the validation score stays constant. The right plot of figure 4 also illustrates this phenomenon nicely.

A study of the regularization parameter λ in figure 5 shows that regularization indeed helps decreasing overfitting, but setting λ to a value greater than 10^{-1} does not improve the validation score further. A similar conclusion can be made for the dropout rate, i.e., the use of dropout indeed helps reducing overfitting. However, the variance of the classifier also increases due to its randomization effect.

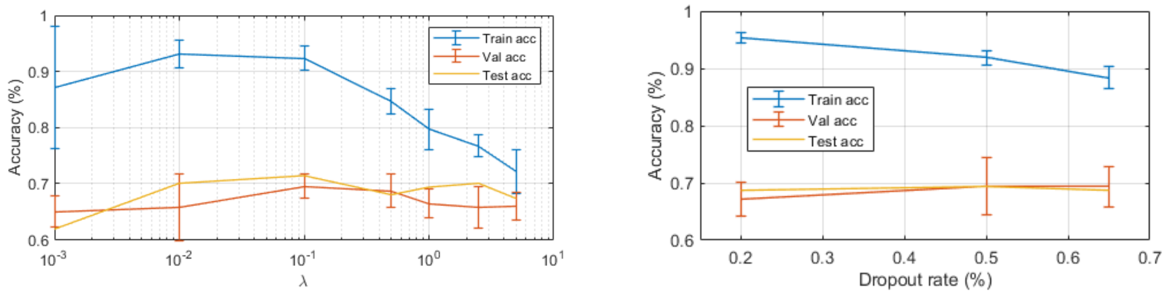


Figure 5: Regularization study: 2-layer ANN

Finally, an important observation can be made regarding the *tf-idf* case: it seems that using *doc2vec* features allows the train network to be more generalizable. In fact, in this case, the gap between the validation accuracy and the test accuracy is negligible, which was not the case in the *tf-idf* case.

4.3 Cases 3 and 4: RNN with tf-idf

Now, different RNN versions are tested based on the features extracted using *tf-idf*, both according to the variable-time-series and constant-time-series schemes (cf. section 3.1). Intermediate experiments to find the optimal embedding size and other hyperparameters are not presented here for the sake of synthesis. The following experiment focuses on the performances of the various RNN model types, with an embedding size of 40, a learning rate of 10^{-4}, a dropout rate of 0.4 and the regularization λ is set to 10^{-2}. In each case, a dense embedding layer has been added before the recurrent unit.

Results show that using 2 hidden recurrent layers slightly increases the network performance compared to the single layer case, but this observation holds for the validation score only. Regarding the generalization to the test set, the best performing network for the variable-time interval case seems to be the 2-layer LSTM, while the Simple RNN performs better in the constant-time interval case. In fact, the latter architecture exhibits a high variance when trained on the variable-time intervals dataset.

Overall, the test accuracy is almost always above the mean validation accuracy, which was not the case for the ANN, leading to the conclusion that the RNN method generalizes better, and

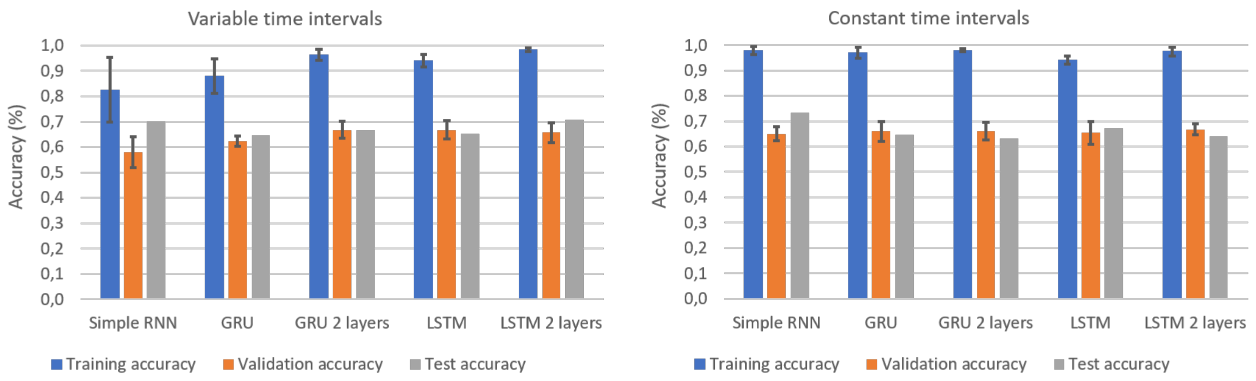


Figure 6: Comparison of the performance of the RNN variants on the *tf-idf* time-series features (K=2500)

highlighting the usefulness of the proposed ensemble method for predicting the test set labels.

4.4 Summary

The following table 4 summarizes the best network configurations obtained for the different feature representation cases:

Features representation	Feature size	Network	Hidden layers	Embedding	λ	Dropout rate	Val Acc mean	Val Acc std	Test Acc
tf-idf no time div.	2500	**ANN**	1	20	0.5	0.5	70.2 %	3.4 %	64.0 %
doc2vec no time div.	2500	**ANN**	2	20	10^{-1}	0.5	69.2 %	3.2 %	68.0 %
tf-idf variable time	2500	**LSTM**	2	40	10^{-2}	0.4	65.7 %	4.0 %	68.8 %
tf-idf constant time	2500	**Simple RNN**	1	40	10^{-2}	0.4	63.6 %	1.21 %	**71.4 %**

Table 4: Summary of the best performing architectures

From these results, it appears that using the constant time division leads to the best testing results and highest network generalizability. Moreover, using features extracted with *doc2vec* also increases the classification performance.

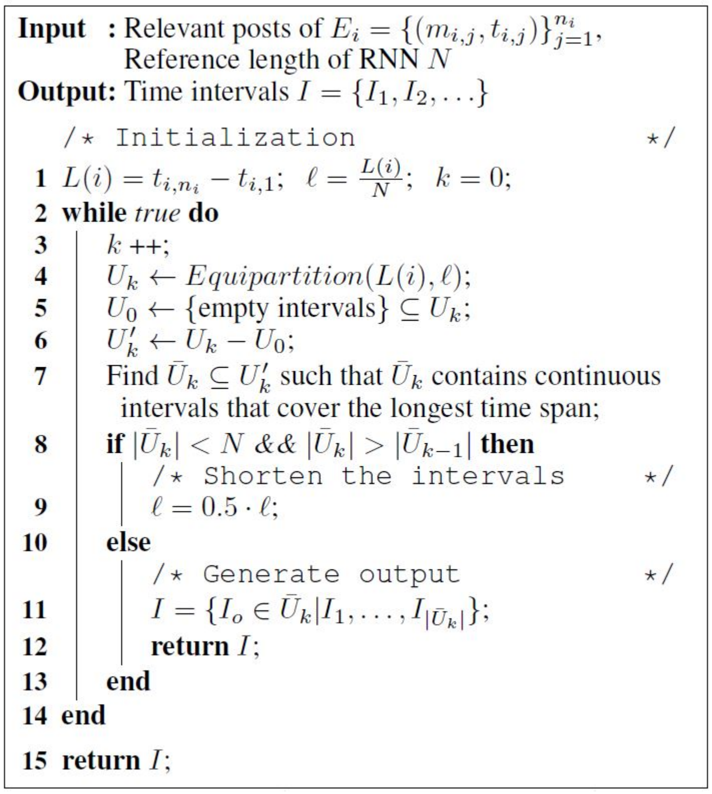
5 Conclusions

In this project, the rumour detection problem has been addressed by gathering a set of tweets for a collection of events, these latter being labeled as rumors or true facts. Two main approaches were considered: first by representing each event as a unique text document and secondly by relying on a temporal division of the tweets. Classification is done using an artificial neural network in the first case, and using a recurrent neural network in the second one, which are trained using the k-fold cross-validation technique to cope with the small dataset size. The actual text features are extracted using two popular methods: *tf-idf* and *doc2vec*.

Results shows that the time-division, namely the constant-time intervals one, leads to higher classification performances (up to 71.4% accuracy on the test set) than the event-wise classification, with a overall increased generalizability of the network. Besides, extracting features using *doc2vec* also leads to better classification results compared to *tf-idf* which has been illustrated in the ANN case.

However, improvements could still be made: the different experiments exhibits high variability and important overfitting. This indicates a strong necessity of increasing the dataset size, which could help refine the experimental results. Additionaly, the *doc2vec* extraction method has not yet been applied to the RNN cases, which could also lead to an improved performance.

A Interval length updating in variable interval approach for RNN



```
Input  : Relevant posts of $E_i = \{(m_{i,j}, t_{i,j})\}_{j=1}^{n_i}$,
         Reference length of RNN $N$
Output: Time intervals $I = \{I_1, I_2, \ldots\}$
  /* Initialization                                   */
  $L(i) = t_{i,n_i} - t_{i,1}$;  $\ell = \frac{L(i)}{N}$;  $k = 0$;
  while true do
      $k$ ++;
      $U_k \leftarrow Equipartition(L(i), \ell)$;
      $U_0 \leftarrow$ {empty intervals} $\subseteq U_k$;
      $U'_k \leftarrow U_k - U_0$;
      Find $\bar{U}_k \subseteq U'_k$ such that $\bar{U}_k$ contains continuous
        intervals that cover the longest time span;
      if $|\bar{U}_k| < N$ && $|\bar{U}_k| > |\bar{U}_{k-1}|$ then
          /* Shorten the intervals                        */
          $\ell = 0.5 \cdot \ell$;
      else
          /* Generate output                              */
          $I = \{I_o \in \bar{U}_k | I_1, \ldots, I_{|\bar{U}_k|}\}$;
          return $I$;
      end
  end
  return $I$;
```

Figure 7: Interval length updating in variable interval approach for RNN [4]

B Word2vec details

There are two approaches in *word2vec* [9], [11], [15], [16]:

- Continuous bag of words (CBOW): this approach is used to predict one word (target word) based on several context words that surround it. The context words are one-hot encoded and fed as input to a Neural Network with one hidden layer and one output layer on which softmax is applied to have a vector of probabilities of possible target words. The input thus consists of C vectors of size $V \times 1$ where V is the number of words in the vocabulary and C is the number of context words. Each of those C vectors is a one-hot encoded word, so a vector with 0s everywhere except at the word index in the vocabulary where there is a 1. The output is one vector of size $V \times 1$. The weights matrix W (illustrated on Figure 8) between the input layer and the hidden layer is thus of size $V \times N$ where N is the number of neurons in the hidden layer. Each row of the matrix thus corresponds to one word, it contains the values used to map this word to the hidden layer, so the values that represent this word. The v-th ($v = 1, ..., V$) row of the matrix W between input layer and hidden layer is thus the feature vector of size $1 \times N$ representing the v-th word in the vocabulary. Thus training the network trains the vector representation of the words in the vocabulary. It is trained by comparing the output vector of probabilities with the one-hot encoded true target word, and by backpropagating the error to adjust

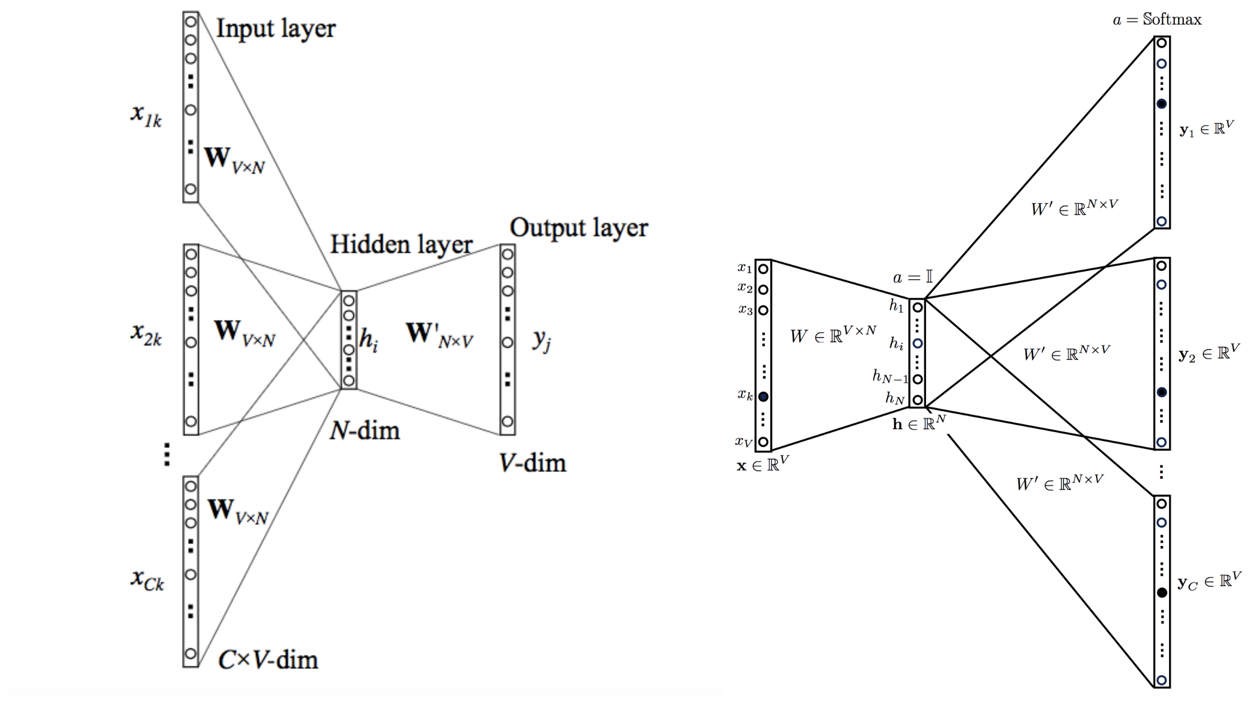

Input layer
x_{1k}
$\mathbf{W}_{V\times N}$
Hidden layer
Output layer
x_{2k}
$\mathbf{W}_{V\times N}$
h_i
$\mathbf{W}'_{N\times V}$
y_j
N-dim
V-dim
$\mathbf{W}_{V\times N}$
x_{Ck}
$C\times V$-dim
$a = \text{Softmax}$
$\mathbf{y}_1 \in \mathbb{R}^V$
$W' \in \mathbb{R}^{N\times V}$
$a = \mathbb{I}$
x_1
x_2
x_3
h_1
$W \in \mathbb{R}^{V\times N}$
h_i
$W' \in \mathbb{R}^{N\times V}$
$\mathbf{y}_2 \in \mathbb{R}^V$
x_k
h_{N-1}
h_N
$\mathbf{h} \in \mathbb{R}^N$
x_V
$\mathbf{x} \in \mathbb{R}^V$
$W' \in \mathbb{R}^{N\times V}$
$\mathbf{y}_C \in \mathbb{R}^V$

Figure 8: CBOW architecture (left) and Skipgram architecture (right) [16]

the weights in W (and W' that is not useful for the word representation). This training process is repeated on all training [target word, context words] pairs, and at the end the rows of W are saved as feature vectors for the corresponding words [11].

- Skip-gram: this approach uses one word to predict the context words. The word is mapped into a hidden layer, that is then mapped to C output vectors, one for each position of the word used for prediction in the context. Again, the rows of the weight matrix form the vector representation of the words [11].

A representation of both approaches is given on Figure 8.

C Data Pre-processing

C.1 Dataset gathering

For each event in the Twitter dataset, the publicly available files only include the tweet identification number (ids), not their content. It is expected of the dataset user to obtain the contents through the Twitter API, however accessing historical tweets older than 7 days through this API requires a Premium license. To get around this problem, the tweets were manually scraped from the Twitter website using a Python script. The following URL format will return the page of the tweet associated with *TweetID*:

https://twitter.com/statuses/*TweetID*

HTTP GET requests with the TweetIDs of the dataset were then sent from the Python script using the *requests* module. The returned data was then parsed using the module *BeautifulSoup*, by finding all the HTML div tags with the class “tweet-text”. The text of the tweet is then the first result. This was repeated from every TweetID in the dataset, and each event was saved as a JSON file containing the event id, the label, and the contents of all the associated tweets. A minimal working example is shown below.

```
import requests
from bs4 import BeautifulSoup

tweet_url = 'https://twitter.com/statuses/1072113227847397376'

headers = {"Accept-Language": "en-US"}
r = requests.get(tweet_url, headers=headers)
soup = BeautifulSoup(r.text, 'html.parser')

tweets = soup.findAll('p', class_='tweet-text')
metadata = soup.findAll('span', class_='metadata')

if len(tweets) > 0:
    #parse date time in US format
    date = metadata[0].text.strip()
    #Save text
    tweets_text = (date, tweets[0].text)
```

C.2 Text cleaning

Prior to feature extraction, the Twitter dataset has to be cleaned. Each JSON file forming the dataset corresponds to a so-called *event*. An event is composed of an original post (i.e. tweet) and all its re-posts (i.e. re-tweets). The cleaning process of each tweet within a JSON file works as follows:

1. All URLs are removed from the tweet, i.e. all strings in the tweet starting with "http".
2. All usernames are removed from the tweets, i.e. all strings in the tweet starting with "@".
3. A combination of the NLTK (Natural Language Toolkit) Python library and of a text cleaning module found on Github called "preprocessor" [17] are used to: remove strings corresponding to smileys, remove hashtags, remove numbers, ensure that all strings start with a lowercase letter, remove punctuation, and stemming.

The stemming process ensures that strings/words which belong to the same category are all mapped to a single string/word. As an example, let us analyse the following list of words: "fishing", "fish" and "fisherman" and "apple". Given that the first 3 words in our example all belong to a single category, they should all be mapped to the same word, for example "fish". Thus after the stemming process, the list of words will be as follows: "fish", "fish", "fish" and "apple". It was decided to use the Porter stemmer from the NLTK library in order to perform stemming.

After the cleaning process has been applied on all tweets from all JSON files of the dataset, the feature extraction process can start.

References

[1] C. Delany, *How Social Media Accelerated Tunisia's Revolution: An Inside View*, The Huffington Post, 2011.

[2] M. Hosenball, *Russia used social media for widespread meddling in U.S. politics: reports*, Reuters, 2018.

[3] R. Moin, Z. Rehman, and K. Mahmood, "Framework for Rumors Detection in Social Media", *International Journal of Advanced Computer Science and Applications*, vol. 9, 2018.

[4] J. Ma, W. Gao, P. Mitra, S. Kwon, and B. Jansen, "Detecting Rumors from Microblogs with Recurrent Neural Networks", *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, 2016.

[5] N. Ruchansky, S. Seo, and Y. Liu, "CSI: A Hybrid Deep Model for Fake News Detection", *CIKM*, 2017.

[6] T. Mayank, *How to process textual data using tf-idf in python*, URL: `https://www.freecodecamp.org/news/how-to-process-textual-data-using-tf-idf-in-python-cd2bbc0a94a3/`. Last visited on 2019/06/05, 2018.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, *Sklearn feature extraction: Tfidfvectorizer*, URL: `https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html`. Last visited on 2019/06/05, 2007-2019.

[8] ——, "Scikit-learn: Machine learning in Python", *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[9] D. Chia, *An implementation guide to word2vec using numpy and google sheets*, URL: `https://towardsdatascience.com/an-implementation-guide-to-word2vec-using-numpy-and-google-sheets-13445eebd281`. Last visited on 2019/06/05, 2018.

[10] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents", *CoRR*, vol. abs/1405.4053, 2014. arXiv: `1405.4053`. [Online]. Available: `http://arxiv.org/abs/1405.4053`.

[11] R. Meyer, *Analysing user comments with doc2vec and machine learning classification*, URL: `https://www.youtube.com/watch?v=zFScws0mb7M`, 2017.

[12] R. Rehurek, *Gensim : Doc2vec paragraph embeddings*, URL: `https://radimrehurek.com/gensim/models/doc2vec.html`. Last visited on 2019/06/06, 2014.

[13] D. Mishra, *Doc2vec gensim tutorial*, URL: `https://medium.com/@mishra.thedeepak/doc2vec-simple-implementation-example-df2afbbfbad5?fbclid=IwAR2eI-A10OVRINN6fvrLoxEj5eVdRrif5kkzMDjT3CRlZJOFKwV5eLXpBw8`. Last visited on 2019/06/06, 2018.

[14] Geoffrey Hinton and Nitish Srivastava and Kevin Swersky, *Lecture notes on Neural networks for Machine Learning. Rmsprop: Divide the gradient by a running average of its recent magnitude*, URL: `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`. Last visited on 2019/06/05, 2012.

[15] S. Huang, *Word2vec and fasttext word embedding with gensim*, URL: `https://towardsdatascience.com/word-embedding-with-word2vec-and-fasttext-a209c1d3e12c`. Last visited on 2019/06/05, 2018.

[16] D. Karani, *Introduction to word embedding and word2vec*, URL: `https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa`. Last visited on 2019/06/05, 2018.

[17] Github, *Preprocessor*, URL: `https://github.com/s/preprocessor?fbclid=IwAR3NRFMNh1qVkcaiFed3GMvWuRmUDNDxz5JGL0--LH58v8vF4upxQdijk2Y`. Last visited on 2019/06/05, 2016.