An Exercise in Automated Fact Checking

Raphaël Piccolin
University College London
Department of Computer Science
London, United Kingdom

ABSTRACT

This project explores the stages of automated fact checking: document retrieval, sentence relevance, claim classification. The document retrieval task consisted in retrieving 5 relevant Wikipedia documents for a given claim. On a sample of 10 claims, we found that retrieval using TF-IDF representations with cosine similarity was the most effective of the methods we attempted. We built a perceptron model for sentence relevance with the help of GloVe word embeddings, obtaining an accuracy of 61 ± 8%. Our simple LSTM model for claim classification had accuracies of 70.06%, while a more advanced model using bidirectional LSTM encoding and attention layers classified 80.51% of claims correctly.

KEYWORDS

document retrieval, recurrent neural network, sentence relevance, claim classification, Zipf's law, LSTM, attention network

1 INTRODUCTION

Fake news detection is quite a popular topic, and has been the focus of some academic research [1][2][4]. With machine learning, we are able to create models which can combat misinformation by automatically performing fact-checking.

In this paper, we are working with the FEVER dataset, which is based on Wikipedia: it provides a large number of claims/queries, and associated evidence sentences from articles. These claims and their evidence are split into three balanced categories: "NOT ENOUGH INFO", "SUPPORTS", and "REFUTES". We ignore the first category to save on computing time and resources.

In the "Procedure" section we discuss Zipf's law in the dataset, various document retrieval and sentence relevance techniques (determining whether an evidence sentence is relevant to a statement). Finally, we propose a preliminary model to classify claims.

The next section, "Literature Review" describes a selection of related works.

Finally, in "Improved: Truthfulness of Claims" we use techniques from the literature review to create a more accurate claim classification model.

2 PROCEDURE

This section of the report details how the code was built, and documents the attempts that lead to the final products.

2.1 Text Statistics

The objective of the first part of this project was to demonstrate that our collection followed Zipf's law. Zipf's law is an empirical law on the frequency of word use in a given language: the frequency at which a word is used is governed by an inverse power law of its frequency ranking. In our case, we fit our data with the following function:

$$f(k;s) = \frac{k^{-s_1}}{\zeta(s_2)} \tag{1}$$

Where k is the rank of a word, s are the parameters of the function which we fit to our data, and ζ is the Riemann Zeta function, used as a normaliser. Typically, Zipf models have only one parameter and a normaliser based on the data size, but we found it more efficient to fit a separate parameter to account for data size. For the 5000 most frequent terms it was found that $s_1 = 0.894978$ and $s_2 = 24.952297$. In figure 1 we show results for the 200 most frequent terms in the collection.

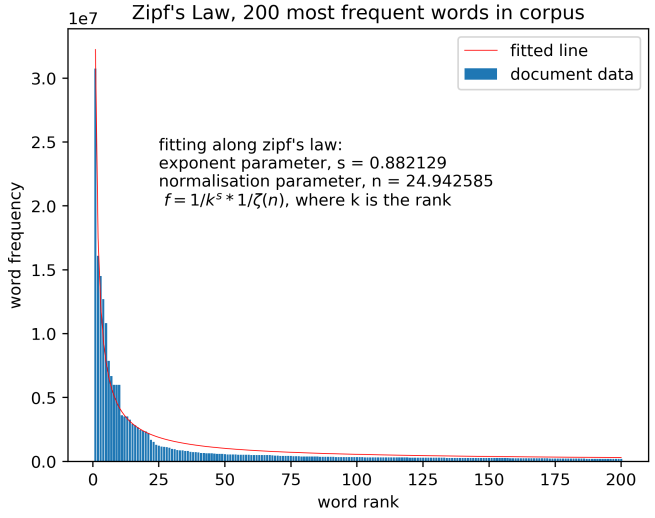

Figure 1: Plot of descending frequency of the 200 most frequent words in corpus along with a fitted line and its parameters.

2.2 Vector Space Document Retrieval

The second subtask was to build Term Frequency - Inverse Document Frequency (TF-IDF) representations of 10 'claims'/'queries' with respect to each of the documents in the collection.

Due to the scale of our collection, it was necessary to build an inverted index of the documents: for each document in the collection we stored its unique ID (title of the Wikipedia article), and a vector. The vector contained the number of occurrences of each unique query word (from our 10 queries). Query words are represented by an index. We encountered an issue with this approach: our index took up too much storage space, making any potential uses very slow. Our database size is approximately 7.5GB, and our inverse index file was 1.5GB. Our solution to this was to replace the array dictionaries which contained the information for each document by an array of space-separated strings (one per document). This string only stores the index of the query words that occurred in the document (the previous index was very sparse) and the number of occurrences. This optimisation allowed us to produce an inverted index of 330MB, which is much more manageable for our purposes.

To build our representations, we defined the TF and IDF of a term as:

$$tf_{t,d} = \Sigma_{t \in q \cap d} t$$
$$idf_t = ln\left(\frac{N}{n_t}\right)$$

Where t is our term of interest, q is the domain of terms in the query, d is the domain of terms a document, N is the number of documents in the collection, and n_t is the number of documents in which the term occurs. To find the best match documents for a query, we proposed the following similarity score, based on the dot product of TF and IDF vectors for the unique terms of a query:

$$sim(q,d) = \Sigma_{t \in q \cap d} \frac{tf_{t,d}}{ln(N_d)} \times idf_t \times \frac{idf_t^2}{\Sigma_{v \in q} idf_v^2} \tag{2}$$

The last term is a normaliser, it divides the squared sum of the IDF of query terms found in a document by the squared sum of IDF of all terms in the query. It was added because it empirically provided better results. Another addition to the standard formula was to divide the term frequency by the natural log of the number of terms in the document, N_d, this is to limit bias towards long documents (not applying the natural log to biases lists where one term is repeated often too much). Finally, we only admitted documents which fulfilled the following condition:

$$idf_t \times \frac{idf_t}{\Sigma_{v \in q} idf_v}$$

i.e. that at a good portion of the key query terms are represented in the document.

Our algorithm, although slow, yielded excellent results, as can be seen in the annex files: all but one of the ten claims were matched with a relevant document.

2.3 Probabilistic Document Retrieval

We employed several variants of a query-likelihood probabilistic document retrieval models. A query-likelihood model estimates the likelihood that a document would produce a query by building a model of the document. That probability is the product of the probabilities of finding each query term in the document:

$$P(Q|M_d) = \Pi_{i=1}^{k} P(q_i|M_d) \tag{3}$$

How $P(q_i M_d)$, the metric we use to rate each document relative to a query, is defined will depend on the model employed.

Basic Model.

$$P(q_i|M_d) = P(w|D) = \frac{tf_{w,d}}{|D|} \tag{4}$$

where |D| is the number of terms in the document. Note that a document which does not contain every query term will have a rating of zero. For short documents this quite problematic and this yielded very efficient but also extremely low recall results (most queries had no non-zero results).

Laplace Smoothing.

$$P(q_i|M_d) = \frac{tf_{w,d} + \epsilon}{|D| + \epsilon|V|} \tag{5}$$

The standard Laplace Smoothing has $\epsilon = 1$, but with the Lindstone correction, we pick an arbitrary small epsilon, in our case $\epsilon = 10^{-4}$ produced good results. $|V|$ is the vocabulary size, or the number of unique words in our collection.

Jelinek-Mercer Smoothing.

$$P(q_i|M_d) = \lambda P(w|D) + (1-\lambda)P(w|C) \tag{6}$$

C represents the collection. λ is a parameter in $[0, 1]$. It is typically chosen in the range $[1, 7]$, closer to 1 for short queries, 7 for longer ones. Its optimal value depends on other several factors as well. $\lambda = 0.4$ was qualitatively determined to be the best value for our query set and collection after experimenting with a range of values.

Dirichlet Smoothing.

This is similar to Jelinek-Mercer Smoothing, except that λ is defined as:

$$\lambda = \frac{|D|}{|D| + \mu}, \quad 1 - \lambda = \frac{\mu}{|D| + \mu} \tag{7}$$

μ is typically the average document length. In our case, $\mu = 85$.

2.4 Sentence Relevance

The aim of this next part is to train a binary logistic regression model which determines whether a sentence is relevant to a claim, using word embeddings of both as an input.

2.4.1 Word Embeddings. Word embeddings consist in turning a word into a vector which is used to compare the usage/meaning of words. To do this we start a sparse word co-occurrence matrix, of size VxV, which represents how many times words co-occur within a context. Then, we can measure word association. One such method is Pointwise Mutual Information, which is a metric comparing how often words co-occur against how often they would be expected to co-occur if they were independent. Applying singular value decomposition makes is possible to represent this information with a denser matrix, of size VxK. Each word is represented by a K length vector.

In our case we imported the $K = 300$ version of Stanford University's GloVe word embeddings. Word embeddings for a sentence were constructed by summing together individual word embeddings and normalising. Normalisation is done so that sentence length is not taken as a factor by the model. For an individual word, embedding vector length tends to represent how common a word is, but direction represents a meaning, so it's preferable to normalise rather than take an average of word vectors to represent a sentence.

2.4.2 Logistic Regression. We implemented a simple, one-layer, binary logistic regression model. The input of the model was an array of length $m = 600$ containing a concatenation of the sentence embedding for a query and for a sentence from a search result document (results from section 2.2).

All the relevant sentence in the search results for each claim were used as positive training data, while random sentences from search results were used for negative training data. The data set

Table 1: Sample of logistic regression results, 2000 epochs

η	avg. accuracy	avg. recall	avg. loss
0.01	60 ± 7 %	36 ± 29 %	0.56
0.15	62 ± 8 %	62 ± 16 %	0.26
0.25	61 ± 8 %	64 ± 15 %	0.19
0.35	61 ± 8 %	64 ± 15 %	0.15
0.4	60 ± 7 %	63 ± 17 %	0.13
0.5	61 ± 8 %	59 ± 15 %	0.12
0.7	59 ± 7 %	63 ± 16 %	0.08
0.9	59 ± 6 %	64 ± 16 %	0.05

was chosen to have approximate class balance. Testing data was retrieved the same way, but using a different set of claims.

The output of the model is given by the following formula:

$$\tilde{y} = \sigma\left(w^\top x + b\right) \tag{8}$$

Where $\sigma = \frac{1}{1+e^{-x}}$ is the sigmoid function, w are the weights, and b is the bias. $Z = w^\top x + b$ is the output of the layer. The sigmoid function is applied to Z in order to return the class probability $p(y_i = 1|x_i)$.

We can calculate the logistic loss of the model at each epoch (or iteration), by comparing the model's output to the correct output, y:

$$J = -\frac{1}{m}\Sigma_{i=1}^{m}\left(y_i ln(\tilde{y_i}) + (1-y_i)ln(1-\tilde{y_i})\right) \tag{9}$$

With backpropagation, we calculate the gradients of the loss with respect to the weights and bias:

$$\frac{dJ}{dw} = \frac{1}{m}x^\top(\tilde{y} - y) \tag{10}$$

$$\frac{dJ}{db} = \Sigma(\tilde{y} - y) \tag{11}$$

Note that here, x is a nxm vector, where n_{train} is the training set size, and y and $\tilde{y}$ are nx1 vectors.

We then update the weights and bias with a learning rate μ:

$$w = w - \eta\frac{dJ}{dw} \tag{12}$$

$$b = b - \eta\frac{dJ}{db} \tag{13}$$

To test our model, we compute $\tilde{y}$ for the testing set.

If $\tilde{y} \geq 0.5$, the model's output is considered positive (i.e. the sentence is relevant to the claim).

If $\tilde{y} < 0.5$, the model's output is considered negative (i.e. the sentence is irrelevant to the claim).

In the case of a binary learning model with class balance for training and testing, it is valid to use accuracy and recall metrics for evaluating the model.

$$accurracy = \frac{\text{true pos. \& neg.}}{\text{\# of retrieved results}} = \frac{\Sigma_{i=1}^{n}|y_i^{ts} - \tilde{y}_i^{ts}|}{n} \tag{14}$$

Using this, we tested combination of model parameters μ and number of iterations to determine which gave the best results (tests were repeated 100 times for each parameter to account for random negative sampling).

Table 2: Relevance evaluation of logistic regression

η	epochs	recall	precision	F1
0.01	8000	41 ± 22%	68 ± 20%	51%
0.1	8000	51 ± 21%	60 ± 18%	55%
0.2	6000	55 ± 21%	58 ± 16%	57%
0.3	4000	56 ± 19%	58 ± 15%	57%
0.4	6000	58 ± 19 %	57 ± 14%	58%
0.5	2000	61 ± 20%	56 ± 14%	59%
0.6	8000	61 ± 20%	56 ± 13%	59%
0.7	4000	62 ± 19%	56 ± 13%	59%
0.8	2000	62 ± 20%	55 ± 14%	59%
0.9	4000	62 ± 20%	55 ± 14%	58%

The data in table 1 shows the evolution of logistic training loss with learning rate. Analysis of this data shows that loss decreases exponentially as learning rate increases.

2.5 Relevance Evaluation

Now, we compute the precision, recall, and F1 score (the harmonic mean of precision and recall) of the sentence relevance model.

$$precision = \frac{\text{true pos.}}{\text{true \& false pos.}} = \frac{\Sigma_{i=1}^{n}\delta_{y_i^{ts},\tilde{y}_i^{ts}} \times y_i^{ts}}{\Sigma_{i=1}^{n}\tilde{y}_i^{ts}} \tag{15}$$

$$recall = \frac{\text{true pos.}}{\text{true pos. \& false neg.}} = \frac{\Sigma_{i=1}^{n}|y_i^{ts} - \tilde{y}_i^{ts}|}{\Sigma_{i=1}^{n}y_i^{ts}} \tag{16}$$

$$F_1 = \left(\alpha\frac{1}{precision} + (1-\alpha)\frac{1}{recall}\right)^{-1} \tag{17}$$

In these equations, the ^{ts} superscript refers to the test dataset (in previous equations y and $\tilde{y}$ referred to the training set), n is the size of the training set.

Selecting parameter $alpha = 0.5$, our logistic regression model obtained F_1 scores of around 59%. The full results are displayed in table 2.

From the various evaluations of our model, we have found that it performs conclusively better than random guessing, although not by a very large margin, and it certainly is not reliable enough to be used to identify relevant sentences. However, performance is in line with expectations, since we only trained our model on a small data size: only 15 relevant sentences (and as many non-relevant sentences) over 10 claims. This was due to limitations regarding computing power.

2.6 Truthfulness of Claims

Building on the previous parts, we can obtain a set of relevant sentences for each claim. In practice however, the relevant sentences for each claim are provided to us by the FEVER dataset, as we have only found these for a small sample of claims in the previous parts. As per the instruction for this assignment, we have filtered out claims which where assigned the “NOT ENOUGH INFO” label, and kept those with the “SUPPORTS” and “REFUTES” label.

The next step in our project is to evaluate the sentences relevant to a claim, to determine if they can be used as evidence to support the claim, or evidence to refute it. We approached this in two different ways, both involving Long Short-Term Memory (LSTM) neural networks. To explore LSTM neural networks further, we shall first explain how an LSTM RNN works.

2.6.1 *Long Short-Term Memory.*

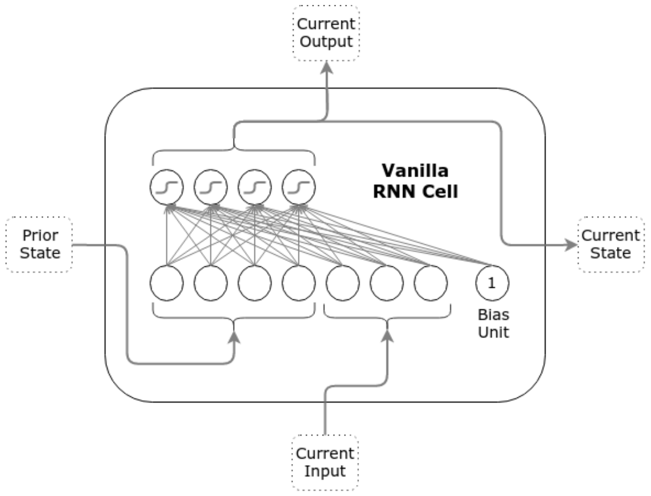

Figure 2: Diagram of a basic LSTM cell [3]. The current input within the sequence is x_t, the current state is h^t, the prior state h_{t-1}, the current output $\hat{y}_t$. The circles correspond to the weight matrices W, U, and V.

A vanilla RNN cell, figure 2, can be summarised to by the following expressions:

$$\hat{y}_t = \sigma(Vh_t + c) \tag{18}$$

$$h_t = tanh(Ux_t + Wh_{t-1} + b) \tag{19}$$

σ is the sigmoid function. $\hat{y}_t$ is the model's output at a timestep t. In our case, we only output at the last timestep. h_t is the hidden layer at timestep t. In an RNN, there is one hidden layer per timestep, where $t \in [0, \tau]$ is the element of a sequence of length τ. U and b are the weight matrix and the bias for the input, respectively. W is the weight matrix for preceding hidden layer. V and c are the weight matrix and the bias for the output, respectively. The weights and biases are shared across all timesteps.

Building on the basic RNN cell, a more complicated example is the LSTM cell (which is a type of RNN cell), shown in figure 3. The main advantage of using LSTM cells over basic RNN is to avoid the 'vanishing/exploding gradients' problem when backpropagating, which occurs in deep nets.

The cell now receives two inputs from the previous cell: its state h_t, as well as the prior memory c_{t-1}. The forget gate's role is to decide whether or not to keep the prior memory, so $f_t \in [0, 1]$. It is defined as:

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \tag{20}$$

The gates i_t and o_t are defined similarly, so we will not write them out explicitly.

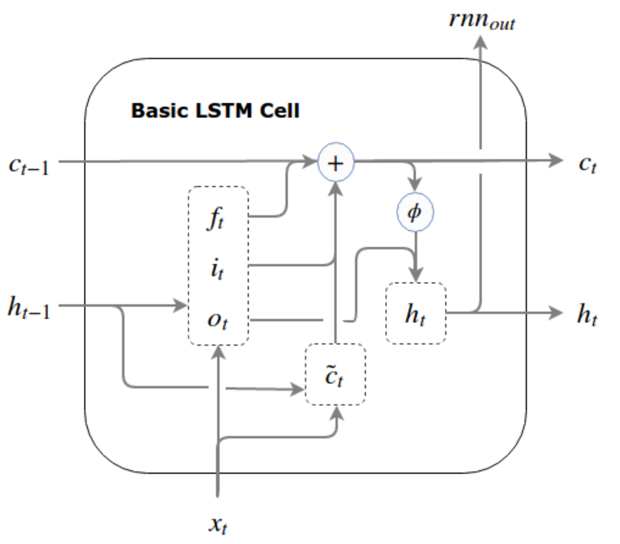

Figure 3: Diagram of a basic RNN cell [3]. c_t is the memory vector at timestep t, with $\tilde{c}_t$ being the current cell's contribution to memory; h_t is the state; x_t the sequential input; f_t, i_t, and o_t are gates with values $\in [0, 1]$; merging lines represent a product; diverting lines represent a copy; + represents an addition; ϕ is an activation function, often tanh.

The input gate, i_t determines whether the current state's memory, $\tilde{c}_t$, should be added to the prior memory.

$$\tilde{c}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \tag{21}$$

Thus, the cell outputs a memory c_t based on the effects of the gates:

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t \tag{22}$$

The output is defined as:

$$h_t = o_t tanh(c_t) \tag{23}$$

Where the output gate o_t determines whether or not to output the memory. Note that in this part we dropped the suffix i in $x_i^{(t)}$ for simplicity. An LSTM network requires a fixed number of timesteps τ for each input $x_i^{(\tau)}$.

2.6.2 *Building a network.*

Figure 4 illustrates how our neural network was built using Keras. We can see examples of both 'many to one' and 'many to many' RNN structures with LSTM cells. The first 'many' in both these layer structures refers to a sequential input, where a timestep is inputted at each cell. In the first case, 'many to one', only the last cell produces an output, while in the second, 'many to many', an output is given at each sequence/cell. The first LSTM layer follows a 'many to many' structure, each cell taking an input $x_i^{(t)}$ and returning the input for the next layer at the same timestep. The final layer is 'many to one' as it needs to provide a single input for the dense layer to produce the final output $\hat{y}_i$. The dense layer is a standard

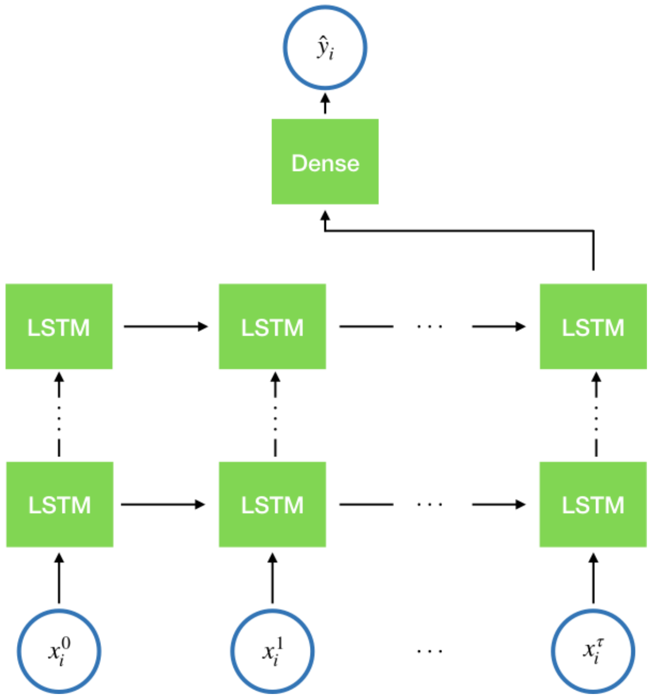

Figure 4: Diagram of our network implementation using LSTM cells.

fully-connected (each neuron receives input from all neurons in the previous layer) layer, which produces a single output using an activation function, in our case the sigmoid function. its output is given by eq. 8.

2.6.3 *Implementations.*

Our first attempt was to treat each sentence separately, so that each input to the network was an array that concatenated the embedding for the claim and for the sentence. So, for each relevant sentence of each claim there was a label indicating whether the sentence refuted or supported the claim. The embeddings were generated with the method from section 2.4.1. Several different architectures were tried, as shown in the file *model performances/task 6-1.rtf*, and they obtained good results. The highest accuracy obtained was 71.83%. Further investigations could include adding more layers and training for a greater number of epochs and lower batch sizes (these all correlate with better accuracy). While these results are satisfactory, they do not exploit the full capabilities of a Recurrent Neural Network (RNN) as this involves minimal sequence programming (using only 1 or 2 timesteps). In effect, when using only one timestep an LSTM network is reduced to a simple multi-layer perceptron, as use of memory and recurence are discarded.

Our second attempt followed the structure described in section 2.6.2. Our inputs were a sequence of sentence embeddings containing the claim embedding as the first member of the sequence, and then successive relevant sentence embeddings. The first issue we encountered was that in our data the number of sentences relevant to a claim can vary. Yet, all sequences inputted to our network must have the same number of timesteps. To counter this, for all inputs which would have less than an arbitrary number of timesteps, we pad them with zero value embeddings. For inputs which would have too large sequences, we remove the excess embeddings.

Our LSTM network uses binary cross entropy to calculate loss, and the Adam optimiser to backpropagate. Our network was implemented using TensorFlow's keras. Results are recorded in *model performances/task 6-2.rtf*. Our best accuracy was 70.06%, but with a slightly better F1 score than with the previous attempts. This suggests we have perhaps not made the best use of LSTM, this will be improved on in section 4.

3 LITERATURE REVIEW

The third section of this report is a literature review on automated fact checking/misinformation detection models. The aim of this section is to provide inspiration for improving the models implemented in this report. The objectives for conducting such a review are as follows:

- Identify the pros and cons of existing methods
- Provide critical analysis
- Find the potential drawbacks for each of the models

Instead of looking at review papers for this literature review, we have chosen to review three papers which propose a neural network architecture for fake news detection (claim verification), and review the various techniques used in these models. Here, we are looking at a particular subset of models which use RNNs. The reason for such a specific selection is so that we can improve on our own LSTM based network from section 2.6. These improvements are to be implemented in section 4.

3.1 SVM with biLSTM embeddings

This first automated fact checking system's [1] objective is to either accept or refute a given claim, working on a rumour detection dataset from *snopes.com*. They used search engines to provide results for queries. First, however, it must generate a short query from a longer claim. The way this is done is by removing all words which are not verbs, nouns, or adjectives (similar to removing stop words). Then, they used an API [1] to identify named entities, and finally ranked terms using a tf-idf representation (idf trained on Wikipedia and Gigaword). A query was generated using the top 5-10 words of each claim, and if a search was unsuccessful the query would be shortened and retried.

Then, document retrieval is achieved by retrieving both the snippet of the result (generated by the search engine) and the document as a whole. One three-sentence extract from the document and one snippet that are most similar to the claim (with three similarity metrics: tf-idf cosine similarity, GloVe embedding cosine similarity, containment) are chosen.

The inputs to the model (top diagram of figure 5) are the output of a bidirectional LSTM layer. It uses pre-trained GloVe embeddings as the sequential input to a biLSTM layer, and outputs a single

[1] www.ibm.com/watson/alchemy- api.html

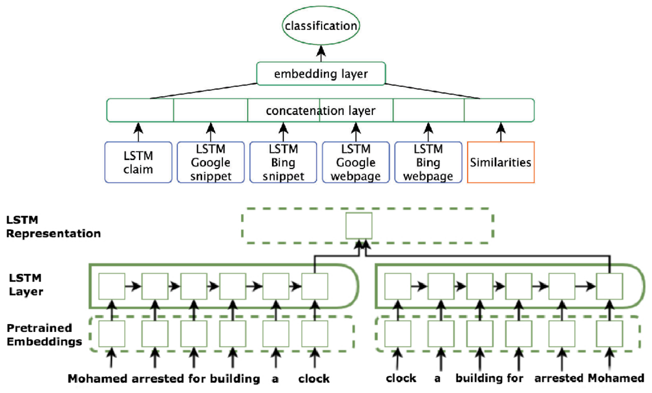

Figure 5: Diagrams of the SVM+LSTM network [1]. Overall network structure (top). Sub-network used to generate LSTM representations of a word embedding (bottom).

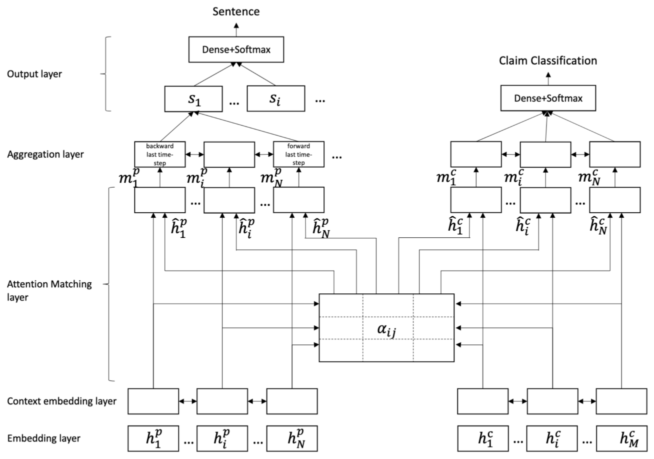

Figure 6: Diagrams of the combined neural network [2].

LSTM representation of the sentence, as can be seen in the bottom diagram of figure 5. Such embeddings are used as an input to the rest of the network. The ‘similarities’ box uses all the same data that was used to produce the LSTM embeddings, but instead simply represents them with an average of their word embeddings.

For comparison, three networks were built based on this:

(1) classification with a simple Support-Vector Machine (SVM) algorithm and a Radial Basis Function (RBF) kernel.
(2) classification with a dense layer and SoftMax output function.
(3) SVM/RBF classification, but without LSTM inputs.

The networks were tested on a dataset which was not class-balanced: 2/3 negative, 1/3 positive labels. Model (3) was the poorest, while model (1) and (2) where similar, although (1) had slightly better results on average. (2) is better at finding true claims, but performs much worse than (1) on false claims, while (1) is quite consistent. The first model is the only one not to overrepresent the majority class, and achieved the best accuracy of the three with 80%.

They also concluded that the choice of search engine did not matter much and that snippets where sufficient to produce reliable results.

Using an LSTM layer to generate sentence embedding is an improvement which can be applied to our project.

3.2 Combined network: retrieval and classification

For the second part of this literature review, we are presenting a paper [2] which uses the same dataset as us (FEVER). The particularity of this paper is that they sought to combine sentence retrieval and claim classification in the same network. They call it an “end-to-end multi-task learning with bi-direction attention” EMBA (model), a diagram of this model is shown in figure 6.

The document retrieval part is achieved using a tool called “S-MART”, which performs entity-linking for Wikipedia, and can retrieve and rank related entities in order to retrieve the top 5 documents for a claim.

The **embedding layer** uses pre-trained GloVe embeddings at both word and character level (separately). The character level embeddings are passed through a Convolutional Neural Network (CNN) with 100 one-dimensional length 5 convolution kernels. Both word and character embeddings are concatenated together and then passed to a “Highway Network” [5]. This is done for both a claim (right on the diagram) and a page (left) so that there is one sequence outputted from this layer on each side of the network.

The **context embedding layer** is a standard bidirectional layer of LSTM cells.

The **attention matching layer** multiplies each sequential output with trained attention coefficients. In effect, this produces a claim-aware representation of each word in a page (left), and a sentence-aware representation of each word in a claim (right).

The **aggregation layer** is another biLSTM layer.

The **output layer** consist of a dense representation followed by SoftMax, with sentence selection on the left-hand side, and claim classification on the right (“SUPPORTS”, “REFUTES”, or “NOT ENOUGH INFORMATION”). The loss for sentence selection is a binary cross-entropy loss. For claim classification, a logit cross-entropy loss is preferred. The overall loss of the network is the weighted sum of both losses (where the weight is a trained parameter).

This method achieved 51.97% accuracy, while the baseline for the dataset was at 52.09%. Other metrics also showed results similar to the baseline. The advantages of this network it performs joint retrieval as well as separate networks would, and that a weak evidence retrieval doesn’t have too much of an effect on claim verification performance: both modules have some degree of independence. However, it performs poorly at determining when claims should be a labelled as “NEI”, and its evidence retrieval results were not encouraging either.

The problems in the methods were:

- S-MART only retrieved the correct documents for 70% of the claims.
- The model only takes the first 800 tokens of each page.
- It does not manage to combine separate evidences together to classify a claim.

While this model is impressive, it is quite difficult to implement, and not particularly effective in terms of final performance. Its use of attention matching and biLSTM layers are particularly relevant to this project.

3.3 Hierarchical attention network

The focus of this last implementation [4] is the use of Hierarchical Attention Networks (HANs). The premise of this paper is slightly different to our task, here the goal is to determine whether a news article itself is truthful. They called their network 3HAN since it uses three encoder/attention layer pairs for embedding, as can be seen in figure 7.

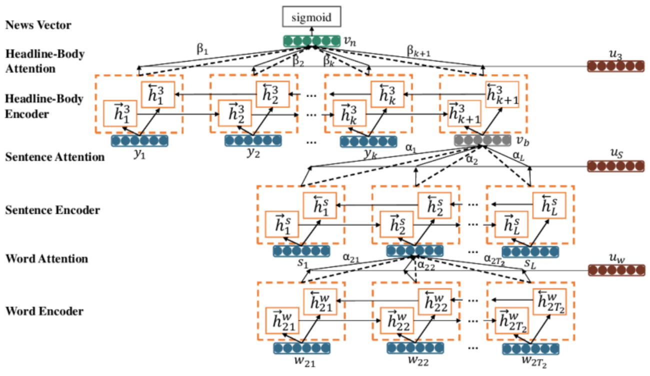

Figure 7: Diagrams of 3HAN [4].

The encoder layers use bidirectional Gated Recurrent Units (GRUs). For the purposes of this project, GRU cells serve functionally the same purpose as LSTM cells. Their input is a sequence of GloVe word embeddings from a text. The attention layers are just standard attention layers (see section 4 for more details on attention).

Every word goes through an encoder/attention pair to produce the embedding of a sentence. This is repeated to obtain an embedding for all sentences in a document. Then, these sentence embeddings goes through a HAN layer to produce a document embedding. In the last HAN layer, the input sequence is the word embeddings for each headline word and the embedding for the document body (the output of the first two HAN layers). This places special emphasis on the headline, reflecting its importance in journalism. Finally, the headline-body embedding goes through a dense layer with sigmoid activation and binary cross-entropy loss. Pre-training is possible for this model, where it is first trained on a 1HAN using only the headlines to obtain better headline embeddings.

This paper compared model efficacy against popular word count based models and neural models. 3HAN performs better, with 96.24% accuracy without pre-training, and 96.77% with pretraining. The best performing word count model is *bag-of-ngrams with TF-IDF* at 92.47% accuracy, and the best neural model is *GRU-ave* (GRU layer with sentence embedding formed of averaged word embeddings) at 95.65% accuracy. 3HAN performed better than similar alternatives which employed word embedding average (3HAN-ave) or max pooling (3HAN-MAX) instead of attention.

It appears that the use of HAN to produce document embeddings is an improvement on the current method implemented in section 4, a similar architecture can be implemented by replacing the headline by the claim.

4 IMPROVED: TRUTHFULNESS OF CLAIMS

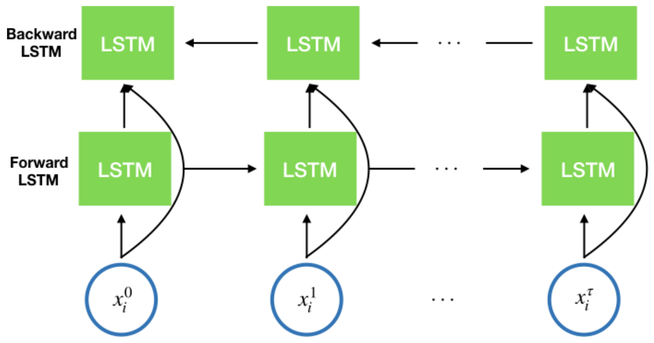

Figure 8: Diagram of a bidirectional LSTM layer. x_i^t represents the t^{th} embedding vector of the i^{th} input line.

Bidirectional LSTM. A first improvement that can be brought to the previous model is replacing forward LSTM layers with bidirectional LSTM (biLSTM) layers, as shown in figure 8. A notable advantage of this layer is that the position of a sequence is no longer as important. With a forward LSTM, the claim might be partially 'forgotten' as it is the first input in the sequence.

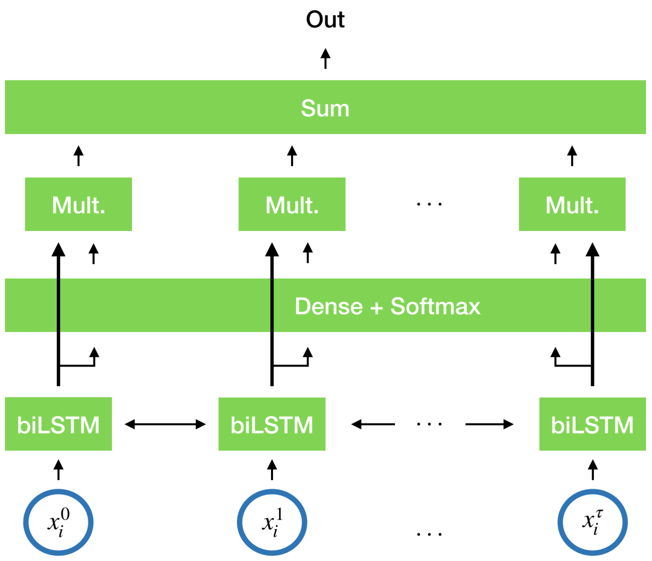

Figure 9: Diagram of an attention model. x_i^t represents the t^{th} embedding vector of the i^{th} input line.

Attention. The next improvement was to add attention to our model. We apply attention on an biLSTM encoding of word embeddings. What this does is attach a weight to each encoding to determine which is most relevant to the claim. The structure is shown in figure 9.

Inputs. The final change to our model was to sample the training data differently. The input to the attention model is the GloVe embeddings of words, we used 50 dimensional embeddings instead of 300 to save on memory/computation. The attention model (with biLSTM) outputs a text embedding, whereas previously we produced sentence embeddings with a normalised sum of word embeddings. Instead of producing encodings for each sentence, we separately generate an encoding for the claim and one for all its evidence. The words of the different evidence sentences are grouped together in the input. We also limited the number of evidence and claim tokens per claim.

Dropout and Masking. Two more subtle improvements to the network where the addition of dropout and masking layers. Dropout is a simple technique designed to prevent overfitting: it consists in randomly setting a portion of input features to zero to prevent overfitting. Looking at the results from the previous model, it is apparent that there is a degree of overfitting, with much better accuracy results on the training data than the test data (see *model performances/task 6-2.rtf*). Masking drops the padding values in a recurrent layer, which improves the performance of the layer. Its purpose is to work around having to input a fixed number of embeddings to biLSTM layers.

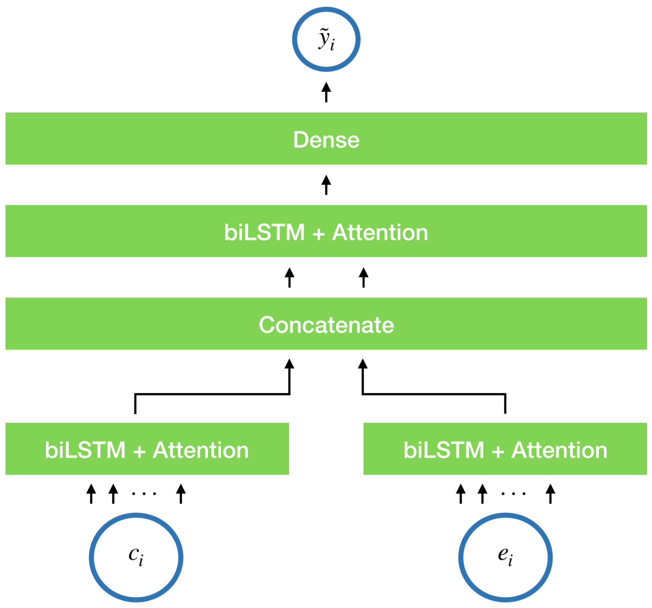

Figure 10: Diagram of the improved model for claim classification. c_i and e_i represents the i^{th} sequence of claim and evidence word embeddings, respectively.

Figure 10 displays a simplified diagram of the improved model, with the "biLSTM + Attention" layer corresponding to that described in figure 9. We are encoding claim and evidence tokens separately at first, and then combining the encodings to generate a 'claim-evidence' encoding, which is then passed through a dense layer for classification. Details of the implementation are readily available in the code section of this project, so we will not mention parameters and settings in this paper. Our results are documented in *model performances/task 8.rtf.* For the final version of the network (network 6 in the results file), 80.51% of claims were classified accurately, a considerable improvement on section 2.6.

4.1 Conclusion

Recapping the results of this project:

- The collection of Wikipedia documents used in the FEVER dataset follow behaviour expected by Zipf's law. The 5000 most frequent words in the corpus can be fitted by
$$f(k;s) = \frac{k^{-0.895}}{\zeta(24.952)}$$
.
- Vector space document retrieval gives more reliable results than the probabilistic document retrieval methods we attempted (of which Dirichlet smoothing was the most effective). However, it requires more expensive calculations. We were only able to test this on 10 claims, which is not sufficient to produce statistically significant results.
- Our standard MLP neural network to evaluate relevance of a sentence to a claim achieved $61 \pm 8\%$ accuracy.
- Our first classification network, using averaged GloVe word embeddings as sentence embeddings and an LSTM layer for encoding, achieved 70.06% accuracy.
- For our second classification network, which feature bidirectional LSTM encoding layers with attention, an accuracy score of 80.51% was obtained.

This project can be used to build an end-to-end fact checking program, by combining the retrieval, relevance, and classification parts of this project together. Issues would include a poor sentence relevance evaluation, although improving on the standard MLP used in this paper should not prove too difficult. Combining different sources could also be a way of improving accuracy, as well as having access to greater computational resources for training.

REFERENCES

[1] Georgi Karadzhov, Preslav Nakov, Lluis Marquez, Alberto Barron-Cedeno, and Ivan Koychev. 2017. Fully Automated Fact Checking Using External Sources. (Oct 2017).
[2] Sizhen Li, Shuai Zhao, Bo Cheng, and Hao Yang. 2018. An End-to-End Multi-task Learning Model for Fact Checking. In *Proceedings of the First Workshop on Fact Extraction and VERification (FEVER)*. Association for Computational Linguistics, Brussels, Belgium, 138–144. https://www.aclweb.org/anthology/W18-5523
[3] R2RT 2016. Written Memories: Understanding, Deriving and Extending the LSTM. Retrieved April 19, 2019 from https://r2rt.com/written-memories-understanding-deriving-and-extending-the-lstm.html
[4] Sneha Singhania, Nigel Fernandez, and Shrisha Rao. 2017. 3HAN: A Deep Neural Network for Fake News Detection. https://doi.org/10.1007/978-3-319-70096-0_59
[5] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. 2015. Highway Networks. *CoRR* abs/1505.00387 (2015). arXiv:1505.00387 http://arxiv.org/abs/1505.00387