Assignment 2 - Fact Checking

GIUSEPPE BOEZIO
Università di Bologna
giuseppe.boezio@studio.unibo.it

SIMONE MONTALI
Università di Bologna
simone.montali@studio.unibo.it

GIUSEPPE MURRO
Università di Bologna
giuseppe.murro@studio.unibo.it

Abstract

Fact checking consists in the task of verifying the reliability of some statement, by comparing it with a given knowledge base. In this experiment, we'll use the FEVER dataset to train a model able to understand whether a fact is verifiable. The model consists in a neural network, structured in different ways, working on embeddings obtained with GloVE. A voting mechanism is then required to make predictions.

1 Introduction

The FEVER dataset contains a high number of claims and evidence (multiple documents per claim), and can be reshaped so that every row of the dataset contains: an ID, a claim, its truth value, a document and whether the document confirms the claim.

2 Dataset processing and model input

2.1 Obtaining the data

As we cannot input natural language data into a neural network, several steps will be needed. First of all, it's required to download the dataset; luckily, this was already implemented, so that the data gets downloaded into 3 CSV files (train, validation, test).

2.2 Preprocessing the data

As it often happens in ML and NLP, the data needs to be cleaned first. After inspecting the dataset, several operations are been considered for both claims and evidences allowing the model to process simpler data, namely making all the text lowercase, substitute characters like brackets and quotes with a space, split all compound words containing a dash, remove all character that are not alphanumeric, period or semicolon. Moreover we noticed that the evidences contain a very raw text, so further cleaning operations are been performed only for the `Evidence` column before the aforementioned ones:

- `remove_beginning_evidence` remove all instances of number and tabulation at beginning of each evidence
- `remove_end_evidence` remove all the keywords after the period at the end of each evidence
- `remove_rounded_brackets` remove all the rounded brackets that are written down as "-LRB-" and "-RRB-"
- `remove_between_squar_brackets` ignore the text between the squared bracket written down as "-LSB-" and "-RSB-" because it contain usually pronunciation spelling or other worthless text

2.3 Embedding

As aforementioned, it's not possible to feed raw text data into a neural network. We can embed the text data using a pre-trained embedding like GloVE. Obviously, some words are going to be missing from the embedding vocabulary, but since these are a minority, we can just compute a random embedding for them and add it to the vocabulary itself. If we had a smaller dataset, we could convert each sentence to the embedding directly. This is though computationally intensive, and it's way better to have a correspondence matrix between a given vocabulary and its embedding, so that we do not repeat data uselessly. The `TextVectorizer` class is responsible of this: it first downloads the GloVE embeddings, then through the method `adapt()`, it processes the embeddings into a dictionary of words and for each of the datasets it adds the Out-Of-Vocabulary words as random vectors and store the `embedding_matrix` as instance variable. These latter operations are wrapped into the method `encode_input()` that is also in charge of transform the claims and evidences in input into sequence of indices with an added 0-padding to handle the majority of our data (99.9% - 88 tokens for the training set). We can then provide this to a `Embedding` layer to achieve the desired input. It is important to notice that using the vectorizer, all the OOV words are computed separately for the train, validation and test set and even though the `Embedding` layer use the last `embedding_matrix` computed, the presence or absence of test OOV words in the matrix will make no difference at training time.

3 Model

The neural network will need to encode two different inputs (claim and evidence), merge them in some way, and output a single value, representing the probability that the claim is correct.

3.1 General model structure

Using TensorFlow's functional API, we're able to instantiate models having multiple inputs/outputs. Abstracting from the implementation choices that will be discussed later, we generally have the following structure:

1. Two inputs, for the claim and the evidence respectively;
2. Masking for these two inputs, ignoring the 0-padding that we added during pre-processing;
3. Embedding for the inputs, using the embedding matrix mentioned in 2.3;
4. Encoding for the inputs, using either a RNN, a dense layer, or just an average;
5. Merging of the encodings, using either concatenation, addition, or averaging;
6. One or more dense layers, finally outputting 1 node.

In the final dense layers, Dropout and regularization techniques are been used to mitigate the overfitting.

3.2 Different encoding strategies

To encode a whole sentence into a 1D vector, we might use different strategies:

- Use a RNN, taking the last hidden state as our input;
- Use a RNN, output the whole sequence (return_sequences=True) and average it;

- Flatten the encoding and insert it into a dense layer;
- Just average the word embeddings.

These strategies have been implemented and can be selected according to the values in the dict `choice_info` passed as parameter to the `build_model` function.

3.3 Different merging strategies

After encoding the two inputs, we need a way of inserting them both into a dense layer. To do this, three type of layers are available in TensorFlow:

- `concatenate`: concatenates the two encodings, obtaining a `[batch_size, 2 * embedding_dim]` input;
- `add`: sum the two encodings, obtaining a `[batch_size, embedding_dim]` input;
- `average`: average the two encodings, obtaining a `[batch_size, embedding_dim]` input.

3.4 Cosine similarity

Information about the similarity of encoding and claim might help the neural network better understand the patterns behind verifiability. Adding this is easy in TensorFlow: as the cosine similarity is just the normalized dot product of the vectors, we can use a `Dot` layer and set its `normalize` argument.

4 Training

All the described models have been tuned using `keras_tuner` obtaining the best configuration using the HyperBand algorithm, maximizing the validation accuracy. The best resulted model has as sentence embedding technique the use of the last state of the LSTM layer whereas the two sentences are combined through an Add layer. The best model extension has been built using this configuration and adding the cosine similarity to the input of the network. In order to avoid the degradation of performance with long training, an `EarlyStopping` callback monitors the validation accuracy (acceptable metric since the targets are equally distributed in the validation set) and stops the training if after 5 epochs no better results are achieved. Figure 1 shows the evolution of the training.

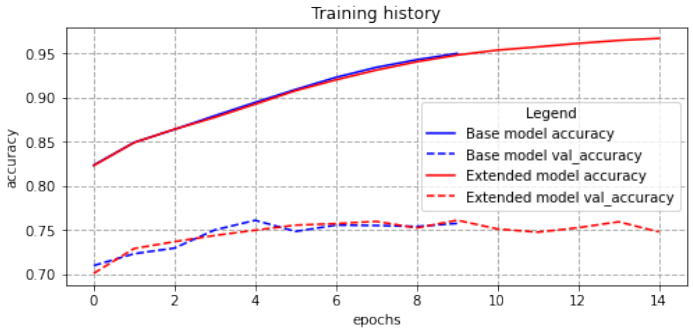

Figure 1: Accuracy of base and extended model during training

5 Evaluation

As aforementioned, the evaluation has been performed after training the models that were found to be the best during the tuning, as seen in 4. Two evaluation strategies have been used: multi-input evaluation (standard approach in classification) and claim evaluation (majority voting).

Set accuracy	base model	extension model
Validation set	0.7609	0.7611
Test set (normal)	0.735	0.743
Test set (majority vote)	0.710	0.718

Table 1: Accuracy of the models on the different sets

As seen in the tables, performances were better for the extended model. In the case of claim evaluation through majority voting it is possible to notice how performances are slightly worse than in the multi input evaluation. This can be explained by the fact that in the test set only the 8% of the claims have more than one evidence.

	precision	recall	f1-score	support
SUPPORTED	0.676	0.907	0.775	3606
REFUTED	0.857	0.562	0.679	3583
macro avg.	0.767	0.735	0.727	7189
weighted avg.	0.766	0.735	0.727	7189

Table 2: Base model performances using multi-input classification evaluation

	precision	recall	f1-score	support
SUPPORTED	0.692	0.879	0.774	3606
REFUTED	0.833	0.606	0.702	3583
macro avg.	0.762	0.743	0.738	7189
weighted avg.	0.762	0.743	0.738	7189

Table 3: Extended model performances using multi-input classification evaluation

	precision	recall	f1-score	support
SUPPORTED	0.642	0.907	0.752	3208
REFUTED	0.857	0.524	0.651	3405
macro avg.	0.750	0.716	0.701	6613
weighted avg.	0.753	0.710	0.700	6613

Table 4: Base model performances using claim verification evaluation

	precision	recall	f1-score	support
SUPPORTED	0.656	0.882	0.752	3208
REFUTED	0.835	0.564	0.674	3405
macro avg.	0.745	0.723	0.713	6613
weighted avg.	0.748	0.718	0.712	6613

Table 5: Extended model performances using claim verification evaluation

The results are satisfactory, though a larger model and a higher quantity of data might push the performances somewhat higher. Encoding the sentences with an attention mechanism, as seen in state of the art papers, could improve the performances too.