DISASTER-RELATED TWEETS DETECTION
Comparing BERT with a custom LSTM

Alessandra Monaco

10-01-2021

Introduction

This notebook detects disaster-related tweets (binary classification) using `pytorch`. The first attempt uses a custom LSTM-based network, while the second one applies transfer learning with BERT network. Both experiments required preprocessing, so phase 1 is applied in any case.

PHASE 1: Preprocessing

The first step involves **preprocessing**: tweets are cleaned and standardized using `nltk` and `regex` as follows:

- removing emojis;
- removing urls;
- removing multiple white spaces and symbols;
- lower casing;
- tokenization;
- stop words removal;
- stemming (Snowball algorithm).

The dataset was then loaded in Torch using `torchtext`, and splitted into **training** (80%) and **test set**(20%); the 70% of the training set was actually used to train the network, while the remaining 30% was used for **validation** during training, to assess the performances of the model. The test set was used, instead, to really evaluate the performances of the network to unseen data, in order to understand if the model is able to generalize.
After preprocessing, some tweets remained empty, because they contained just non-alphanumeric characters, so they have been removed from the dataset.

CUSTOM LSTM NETWORK

PHASE 2: Featurization

The preprocessed and tokenized tweets were **featurized**, meaning that the lists of tokens were converted into vectorial representations. There exist many representation that are suitable for text data, as Bag Of Words with binary coefficients, Count Vectorizers,Tf-idf coefficients, Word Embeddings. This last one transforms each word into a vector, such that the representation is able to capture the context of the text, because words of the same context usually appear together in the corpus, so they will be close in the vector space as well. In some sense, this representation is able to capture the *meaning* of the words through the vector, and this is why this featurization was preferred to the standard tf-idf.

The vocabulary was built using just the training set, and the representation was learned through the **Embedding layer** of the network. We ended up with 12 102 word-vectors, each with 200 dimensions; 100 and 150 dimensions were tried too, giving worse results. Moreover, we initialized the vectors with the pretrained glove vocabulary with 100 dimensions (`glove.6B.100d`), and the twitter glove with 200 dimensions (`glove.twitter.27B.200d`).

PHASE 3: Model Architecture

Building a Neural Network can be challenging, and involves taking many decisions: what kind of layers should be added? How many layers (depth)? How many neurons at each layer (width)? What kind of loss function? What kind of optimizer? Generally, there are no clear and fixed to rules to deal with that; the choices may depend on the dataset and may be just experimental. Anyway, there is some knowledge and guidelines that can be exploited, for instance it is well known that LSTM layers perform well with text classification, because they are able to deal with sequences of input, in this case, sequences of words.

Our neural network development was experimental and step-wise: we started with a very basic network with an Embedding layer, LSTM layer, fully connected layer and sigmoid activation. It is known in fact, that a sufficiently wide network is able to approximate any function. We trained and tested the network, starting to add more layers and changing the dimensions, and therefore, the number of parameters.

The final architecture is the following:

```
Net(
  (embedding): Embedding(12102, 200)
  (lstm): LSTM(200, 64, num_layers=2, batch_first=True, dropout=0.3, bidirectional=True)
  (fc): Linear(in_features=128, out_features=50, bias=True)
  (dropout): Dropout(p=0.3, inplace=False)
  (fc2): Linear(in_features=50, out_features=1, bias=True)
  (act): Sigmoid()
)
```

The **Embedding layer** learns the word embeddings, creating the word vectors.
The **Lstm layer** has a complex structure based on hidden state to memorize; using the bidirectionality, we enable the layer to preserve information from both past and future in the sequence, such that the context can be understood better.
The **Linear layer** is the fully connected one, applying a linear combination of weights and input features (128).
Between the two linear layers we added a **Dropout layer**: this basically disable some neurons at random (these neurons can not learn), to prevent overfitting of the data. Of course, this must be carefully tuned: if it is too small we may risk overfitting, if it is too high we may make the network incapable of learning anything. The dropout is addedd also in the Lstm layer for the same reasons.
The activation function is a **sigmoid** because we are dealing with binary classification: the output of a sigmoid is in [0,1], and it is interpreted as a probability[1]

PHASE 4: Model training

We defined suitable functions to train our model (`train`, `evaluate`, `train_early_stop`).
The **Early Stopping** is another solution to prevent the network training too much, and therefore, overfitting. We keep track of the best loss found so far, and at each epoch the current validation loss is compared with the best one; if we do not see any improvement after a certain, predefined number of epochs (`patience`, the training stops, and the best model is stored in a file.
We trained for 100 epochs with a patience of 12, but any training attempt required at most 30 epochs to converge.

The optimizer used was **Adam**, since it is known to be state-of-the-art, due to its capability to adapt learning rates using the second moments of the gradients. The starting **learning rate** was 0.0005: having larger learning rates in fact, made the network incapable of learning, outputting always the same result. This happened because a large update on weights caused the model to converge too quickly to a suboptimal solution.

Regarding the **loss function**, the binary classification requires a Binary Cross Entropy (BCE) loss. Anyway, this function in PyTorch does not allow to define positive and negative weights, a functionality that is almost necessary when dealing with *imbalanced data*. Our training set in fact, contained 5206 samples of class 0 ("non-disaster" tweet), and 1157 samples of class 1 ("disaster-related" tweet). Many solutions and algorithms deal with imbalanced data. *Undersampling*, for instance, balances the dataset removing samples from the majority class; *oversampling*, at the opposite, balances the dataset adding samples to the minority class, using SMOTE, or with augmentation. This last solution was experimented in this notebook (just on the training set), using `nlpaug` library to generate new tweets for disasters using `ContextualWordEmbsAug`, an augmenter based on word embeddings. Anyway, the attempt failed, probably because the augmentation introduced too much noise in the data.
Thus, our choice was to give a different weight to the classes in the loss function, such that errors made in predictions on the minority class are considered more relevant (high weight) than errors on the other class, improving training. A good practice is to set `pos_weight` to the ratio between the size of the majority class and the size of the minority class.
The weights were passed as arguments to the **BCEWithLogitsLoss** function.

The network is trained in batches of 16 samples (`batch_size=16` showed better performances than 32 and 64).

[1] if output>=0.5 then first class; if output<0.5 then second class.

PHASE 5: Model testing and results

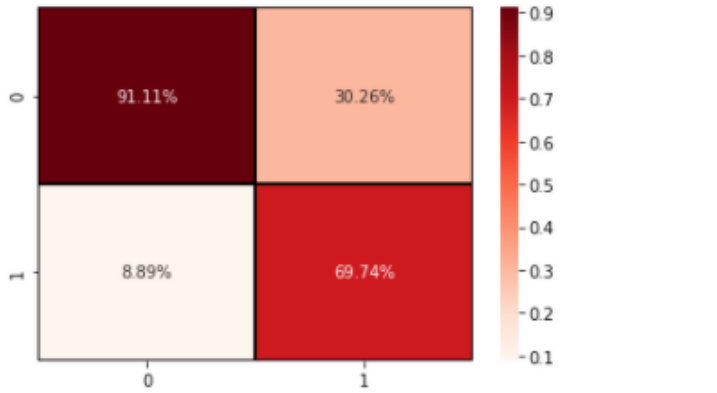
Tuning hyperparameters we found our best model:

```
#define hyperparameters
size_of_vocab = len(TEXT.vocab)
embedding_dim = 200
num_hidden_nodes = 64
num_hidden2_nodes = 50
num_output_nodes = 1
num_layers = 2
bidirection = True
dropout = 0.3 #inner dropout (LSTM)
dropout_layer = 0.3
lr = 0.0005
```

with these validation metrics:

```
CLASSIFICATION REPORT

              precision    recall  f1-score   support

           0       0.91      0.94      0.92      2201
           1       0.70      0.62      0.66       526

    accuracy                           0.87      2727
   macro avg       0.80      0.78      0.79      2727
weighted avg       0.87      0.87      0.87      2727
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f9d5d8863c8>
```



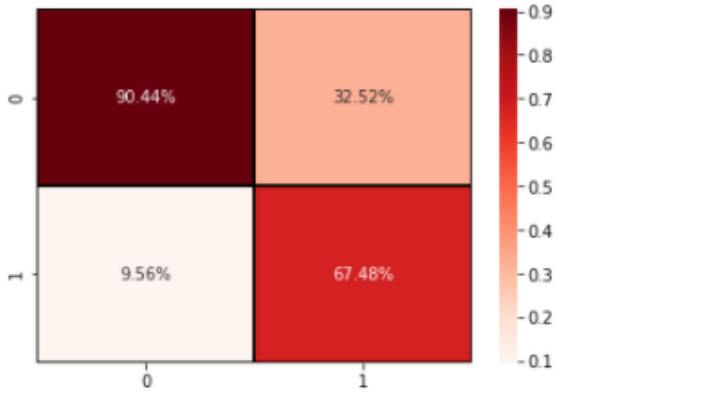
Expecially with imbalanced datasets it is very important to check other metrics than accuracy: accuracy indeed takes just into account the amount of correctly classified instances (*true positives, true negatives* and wrongly classified instances (*false positives, false negatives*, therefore, we can have a very high accuracy even if the model is misclassifing most of the minority class instances. We wish, instead, to have a model that performs quite good with both classes, not jut with one. The **confusion matrix** helps to visually understand the performances on the different classes; in this case we can see that the class 0 (non-disaster) is well predicted (91% of the samples are correctly classified), and the predictions of class 1 are almost acceptable (70% of the disaster-related samples are detected, while 30% are false negatives).

Validation set performances are not enough to understand how well the model is predicting; we want to really understand how it deals with unseen data (validation set is used during training to evaluate each epoch), therefore the model is used to predict on the test set, with the following results:

CLASSIFICATION REPORT

	precision	recall	f1-score	support
0	0.90	0.93	0.92	1841
1	0.67	0.58	0.62	431
accuracy			0.87	2272
macro avg	0.79	0.76	0.77	2272
weighted avg	0.86	0.87	0.86	2272

<matplotlib.axes._subplots.AxesSubplot at 0x7f9d5cd61be0>



The performances are quite similar to the validation, so the network is almost able to generalize.
We observed that, without weighting the loss function, the precision, recall and f-score were bad, even with a good accuracy, due to the imbalancing of the dataset; in particular, the precision of the two classes were similar (0.87 and 0.78), but there was a huge difference between the recall of class 0 (0.97), and the recall of class 1 (0.39), and therefore the percentage of false negatives in the confusion matrix was too high.
For our purposes (detect disaster tweets to improve emergency awareness), we want to increase the recall, the portion of actual positives that is correctly identified. False positives, infact, are less important than false negatives, because a false positive is just a false alarm, while a false negative implies that we are loosing information about a disaster that is happening in the world, failing the task (detect disasters to provide fast help).

With the weighted loss, we were able to increase the recall to 0.58, but we had to "sacrifice" some precision on the disaster class. Moreover, we observed that without the weighted loss the difference between the training loss and the validation loss was high, increasing after epoch 2, and this could be a sign of overfitting. Using weights instead, the loss values were very similar (0.8 and 0.9).

During the tuning, as stated before, we changed also the depth and the width of the network, observing that the wider network tended to overfit (we are just introducing more parameters to be learned), while the deeper network improved the performances and the generalization capability. This happens because, stacking more layers, the network can learn features at various levels of abstractions, learning a more complex function. Increasing the depth of the network helped to learn intermediate features, even with our small dataset.

Regarding the featurization, we observed that, with the "first" network (just Embedding, Lstm, Linear and sigmoid), there was an improvement on the performances initializing the vectors with twitter glove; using the deeper network instead, the performances were very similar to each other.

TRANSFER LEARNING (FINE TUNING) WITH BERT

Using the same preprocessing (PHASE 1), we applied Fine Tuning, taking the famous pretrained network BERT, and using its weights as initialization for the training on our dataset.

PHASE 2: Featurization

BERT featurization is a bit more complex: BERT network takes as input a particular **Bert Object**, of the form (text, id, mask, segment_id, label_id) for each tweet. This transformation is accomplished by the function `convert_examples_to_inputs`, and it is done for training, validation and test sets.

PHASE 3: Model import

After featurization, we imported the BERT model and the BERT tokenizer. Both `bert-base-uncased` and `bert-large-uncased` were tried. BERT base has the following parameters: L=12, H=768, A=12 where L is the number of stacked encoders, H is

the hidden size and A is the number of heads in the MultiHead Attention layers. BERT large is basically larger and more compute-intensive: L=24, H=1024, A=16 (the training was indeed slower).

PHASE 4: Model training

The dataset in Bert format was passed to the network using data loaders, with batches of 16. This batch size is the maximum allowed by our system to avoid Cuda out of memory for GPU. The model was trained with **Early Stopping**, using a patience of 6, with a learning rate of 0.00005. We needed just a pair of epochs to get the best model, probably due to the fact that the model was pretrained.

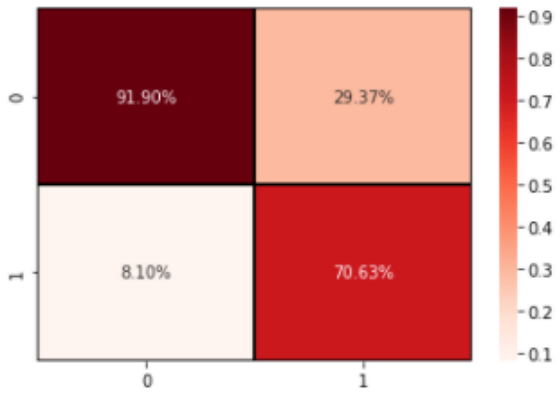
PHASE 5: Model testing and results

The best model was found using `bert-large-uncased`, but it took some time to be trained.
Training performances (accuracy) are shown below:

```
Training performance: (0.964010686782964, 0.964010686782964, 0.964010686782964, None)
Development performance: (0.8925559222588926, 0.8925559222588926, 0.8925559222588926, None)
```

As we did for the Custom Lstm, we tested Bert on test set, with the following results:

```
Test performance: (0.8820422535211268, 0.8820422535211268, 0.8820422535211269, None)
Accuracy on test set:  0.8820422535211268
              precision    recall  f1-score   support

           0       0.92      0.94      0.93      1841
           1       0.71      0.65      0.68       431

    accuracy                           0.88      2272
   macro avg       0.81      0.79      0.80      2272
weighted avg       0.88      0.88      0.88      2272
```



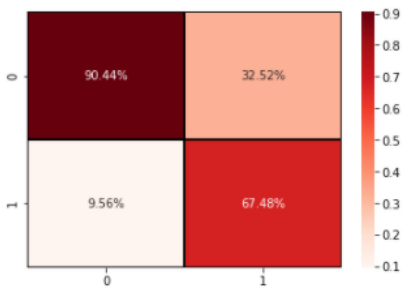
Models comparison and final notes

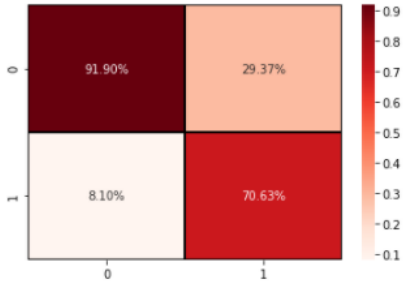
We show again the performances of the two models, to make comparison easier:

```
CLASSIFICATION REPORT
              precision    recall  f1-score   support

           0       0.90      0.93      0.92      1841
           1       0.67      0.58      0.62       431

    accuracy                           0.87      2272
   macro avg       0.79      0.76      0.77      2272
weighted avg       0.86      0.87      0.86      2272
```

```
Test performance: (0.8820422535211268, 0.8820422535211268, 0.8820422535211269, None)
Accuracy on test set:  0.8820422535211268
              precision    recall  f1-score   support

           0       0.92      0.94      0.93      1841
           1       0.71      0.65      0.68       431

    accuracy                           0.88      2272
   macro avg       0.81      0.79      0.80      2272
weighted avg       0.88      0.88      0.88      2272
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f9d5cd61be0>
```





Results show that Bert is better to recognize disaster tweets (class 1). The precision and the recall (and consequently the

f-score) are higher, and the portion of misclassified disaster tweets is now about 29%.
This is basically due to the fact that Bert architecture has many improvements:

- Bert is deeper (about 23 Bert layers); most of the time, a deeper network may help to learn more features, learning a complex function;
- Bert layers include the attention mechanism: this enables the network to understand important features;
- the featurization is more sophisticated: the advantage of using a transformer instead of a word embedding is that the vector assigned to a word is a function of the entire sentence, therefore a word can have different vectors based on the contexts, supporting polysemy;
- it uses warm-up to reduce the primacy effect of the early training examples;

Moreover, we noticed that, training the same Custom network multiple times, the results (expecially for class 1) change more often than training Bert multiple times: our hypothesis is that some samples of class 1 are still predicted at random. With Bert, results are more stable, probably because the featurization and the attention enable the network to learn better the class.