Textual Entailment

NLP Course Project

By: Yehuda Yadid & Eliran Shem Tov

Lecturer: Ella Rabinovitch

Table of Contents

Textual Entailment – NLP Course Project

By Yehuda Yadid and Eliran Shem Tov

Introduction

The Task

As part of our Natural Language Processing course, we were instructed to use the knowledge we've gained and create a software that will recognize textual entailment (sometimes referred as Inference) and based on the *Stanford Natural Language Inference* (SNLI) *corpus.*
Recognizing Textual Entailment (RTE) is a relatively advanced task that aims to classify directional relation between texts. There are two flavors of the problem: 2-way and 3-way classification.

Definitions - For Text T' and Hypothesis H', we'll say that:

- 3-way classification:
  - T entails H if humans reading of T will infer that H is most likely true.
  - T contradicts if humans reading of T will infer that H is most likely false.
  - T neutral if humans reading of T will infer that H is most undetermined.
- 2-way classification:
  - T entails H if humans reading of T will infer that H is most likely true.
  - T doesn't entail H if humans reading of T will infer that H is most likely false.

Ambiguity, variability, and the complexity of natural languages makes expressions and meanings get mapped by a many-to-many relationship. The RTE problem is a relaxed variation of the paraphrasing problem, and it offers a wide variety of applications such as information extraction, summarization, question answering and more. To achieve good results in our task, our program should have the ability to semantically understand the text at some level.

Our Solution

During our work we went through a lot of ideas. Some better, some less, few got implemented and others were dropped in theory phase. Eventually the purposed solution we came with is to use the BERT model that we mentioned in class and retrained it on our corpus for fine-tuning and adjustments to excel the RTE task.

BERT (Bidirectional Encoder Representations from Transformers)

A transformer is a deep learning model that uses a self-attention among other parameters to adjust its weights. Unlike RNNs and CNNs which does so as well, transformers don't apply constraints on the order of data being processed, fact that allows greater parallelization. BERT is a transformer-based model which was developed by google (2018) and summoned a new era in the research of NLP. Today it is being used widely in research as well as in production of many applications.

There are two published models of BERT: *BASE* (12 encoders with 12 bidirectional self-attention heads) and *LARGE* (24 / 16). Both are pre-trained on unlabeled data taken from Wikipedia (2,500M words) and from the BooksCorpus (800M words).
The pre-training phase is expensive in both time and resources, and as such done once. After this step, the model can be fine-tuned using less resources, based on smaller datasets, to achieve good results for other specific tasks (other than the original language modeling and sentence prediction tasks that the model had). As part of its pre-training phase, which is bi-directional per input, BERT learns and generates contextual **word embedding**.

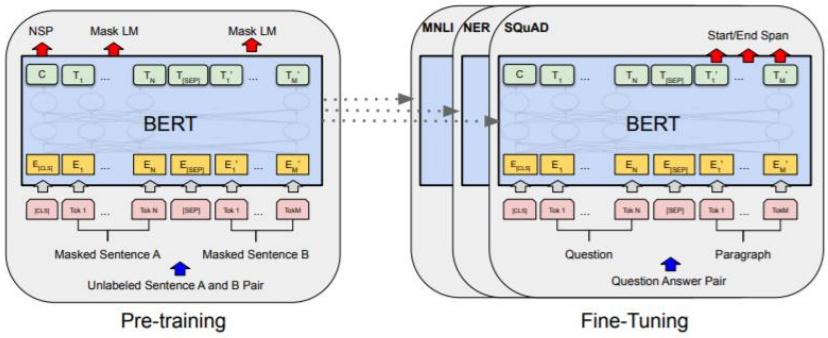

Figure 1: Overall pre-training and fine-tuning procedures for BERT. Apart from output layers, the same architectures are used in both pre-training and fine-tuning. The same pre-trained model parameters are used to initialize models for different down-stream tasks. During fine-tuning, all parameters are fine-tuned. [CLS] is a special symbol added in front of every input example, and [SEP] is a special separator token (e.g. separating questions/answers).

Word Embedding

The process of generating numerical representation for a word is called word embedding. This numerical representation is usually a dense vector of real numbers that encodes the word's meaning. The meaning is simulated as embeddings such that similar words will be closer in the vector space. During the course we've mentioned few word embedding techniques such as the naivest One Hot Encoding, TF-IDF as an occurrence counting improvement that yields good results for many tasks, word2vec that learns lexical semantics of a word using its context (predicting a word by context or predicting a context by word) and more. The BERT model generates such word embedding as well. As a matter of fact, the model's researchers and creators already embedded a huge vocabulary in the pre-training phase mentioned above.

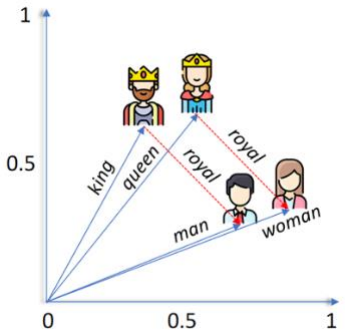

Figure 2 | Illustration of words in the vector space

Once having a vectorized representation of the data, that can teach us the semantical meaning of every word by its location in the space, we can benefit from the spatial properties of embedding and apply arithmetic operations and calculate the similarity or distance between words or sequences.

The Data

Our RTE task is based on the SNLI corpus (v1.0) which is a large collection of 570K English sentence pairs that were labeled manually with the labels: [entailment, contradiction, neutral]. The best classification results for RTE with the SNLI corpus were achieved by Pilault et al. ('20) with their CA-MTL model (92.1% accuracy).

Every instance in the corpus is composed of premise (T), hypothesis (H) and label. In fact, there are some other attributes for every instance, but we'll use "sentence1", "sentence2" and "gold_label" solely.

```
{
'premise': 'Two women are embracing while holding to go packages.'
'hypothesis': 'The sisters are hugging goodbye while holding to go packages after
 just eating lunch.'
'label': 1
}
```

Figure 3 | Simplified example instance from SNLI corpus

The data is divided to datasets of the following instances sizes: Train - 550,152 | validation - 10,000 | test - 10,000.

The labels of the train set instances are almost equally distributed, where each label appears in ~33.3% of the set. On our pre-processing phase we dropped the instances with missing gold-labels (0.1% of the dataset). The median number of tokens in the premises of the train set is 12, while this value of hypothesizes is 7.

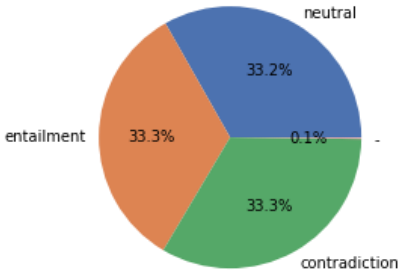

Figure 4 | The distribution of gold labels in the SNLI training set

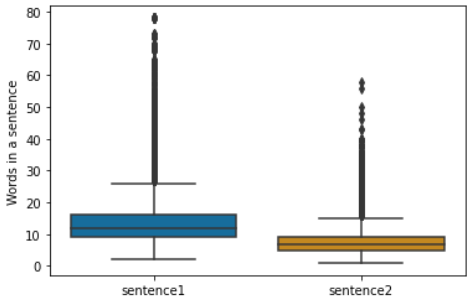

Figure 5 | Premises & Hypothesizes Lengths Distribution

index	gold_label	sentence1_binary_parse	sentence2_binary_parse	sentence1_parse	sentence2_parse	sentence1	sentence2	captionID	pairID	label1	label2	label3	label4	label5
count	550152	550152	550146	550152	550152	550152	550146	550152	550152	550152	39370	39395	39383	36914
unique	4	150736	479906	150736	479908	150736	480040	151196	550152	3	3	3	3	3
top	entailment	(((A dog) (in (a field))) .)	((A man) ((is sleeping)))	(ROOT (NP (NP (DT A) (NN dog)) (PP (IN in) (NP (DT a) (NN field))) (. .)))	(ROOT (S (NP (DT A) (NN man)) (VP (VBZ is) (VP (VBG sleeping))) (. .)))	A dog in a field.	A man is sleeping.	2978540629.jpg#1	518230621.jpg#2r1n	entailment	contradiction	contradiction	contradiction	entailment
freq	183416	33	335	33	335	33	335	15	1	183384	13478	13728	13701	12768

Table 1 | SNLI Training set description

Solution

Dive into our solution

As mentioned earlier, our purposed solution is based on the BERT model. We used Facebook's open source PyTorch framework and hugging-face transformers library and implemented the solution in a JuPyter notebook. We initially played around with the free Kaggle's NVidia K80 GPU kernel. When we got to the real action part of training the model twice (3-way and 2-way classifications) we switched to a paid Google Cloud N1-Standard-4 instance with NVIDIA Tesla T4x1 GPU.

The model was pre-trained ahead and already holds word embedding for language modeling and prediction tasks for which the model was trained for. However, our task is not one of the pre-training tasks and therefore we are required to adjust the model parameters for our needs. To achieve the desired fine-tuning of the model weights and parameters, we need to preprocess the RTE task specific dataset and train the model on it.

The preprocessing phase is concluded of the following steps:

1. Loading the data as pandas' data-frames and use only the *gold_label* as label, *sentence1* as T' and *sentence2* as H' columns.
2. General cleanup – removing pairs with missing H' value & removing instances without a gold label.
3. Initializing a BERT tokenizer based on the base model (12 encoders + 12 bidirectional self-attention heads).
4. For every <T', H'> pair, generate a unified sentence with BERT's expected separators: [CLS]T'[SEP]H'[SEP]
5. Tokenize the unified sentence (3) using the tokenizer generated at (2).
6. Prepare a pair of sequences (T' & H') as an input for the model:
    a. Pad sentences to match the longest sentence to maintain rectangular shape of the model's input. We could have shortened all to the median length for example but then we would lose data or setting a bigger length but then the training time would have grown unnecessarily.
    b. Applying an attention mask to tell the model to ignore the padding values.
    c. Convert sentences to sequence of ids (of vectors known to the model)
7. Apply the above (1-6) on the train, validation, and test datasets, and save the preprocessing results as python pickles so this process could be done once.
8. Generate a PyTorch tensor datasets and create a data-loader for each.
9. Move the model and the processed data to the GPU device, for accelerated results.

We specifically used *BertForSequenceClassification* with the 12 encoder layers and its full structure is detailed in the notebook. In addition, we used AdamW optimizer (with a learning rate = 2e-5). Then we re-trained the model, with only 3 epochs due to time constraints and research that we've seen that demonstrated good results with this number of epochs.

Results

Accuracy & Timing

Re-training the model on our processed SNLI corpus took 3 hours for each epoch (~9 total) and achieved 0.89% accuracy for the 3-way classification, and 91% accuracy for the 2-way task (with similar timing).

Loss Validation & Over Fitting Avoiding

Over fitting occurs when the model fits too good to the training data. When such fitting takes place, the model won't be able to perform accurately against new, unseen data. Therefore, we want to make sure that we can see a correlation between the accuracy and loss values of both the train and validation sets. Since we used only 3 epochs, we cannot learn much from the charts bellow, but we can see a positive trend.

3-Way Classification

	epoch	training_loss	training_accuracy	validation_loss	validation_accuracy
0	0	0.412927	0.842313	0.298896	0.890794
1	1	0.292383	0.894614	0.277398	0.901019
2	2	0.232486	0.918586	0.325372	0.892925

Table 2 | 3-Way loss and accuracy over epochs

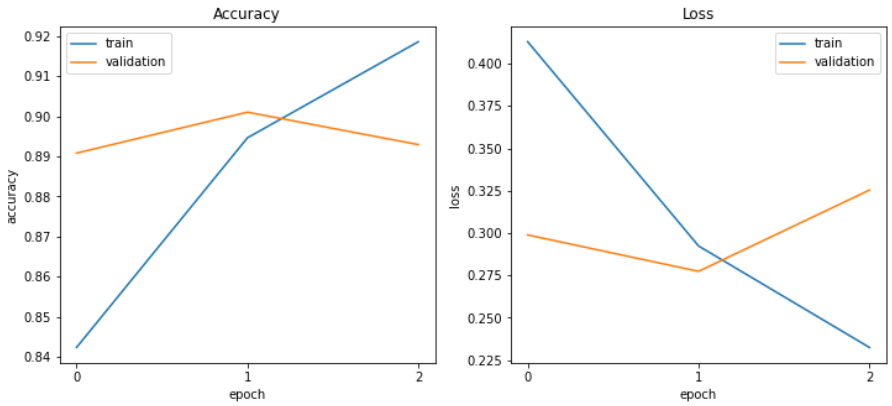

Figure 6 | Training & validation accuracy and loss over epochs in 3-way classification

2-Way Classification

	epoch	training_loss	training_accuracy	validation_loss	validation_accuracy
0	0	0.234071	0.907563	0.172214	0.932776
1	1	0.174190	0.934723	0.162952	0.943328
2	2	0.138535	0.949514	0.168336	0.938966

Table 3 | 2-Way loss and accuracy over epochs

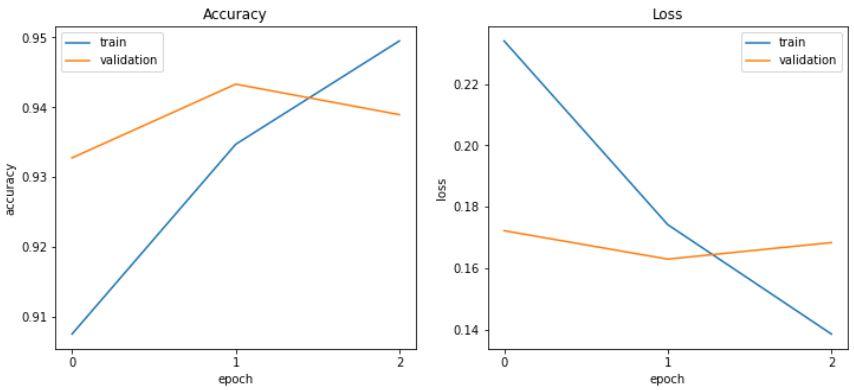

Figure 7 | Training & validation accuracy and loss over epochs in 2-way classification

Classification Report

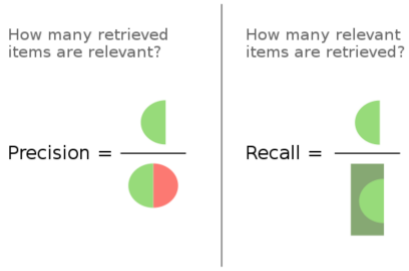

Figure 8 | precision and recall

	precision	recall	f1-score	support
0	0.86	0.86	0.86	3229
1	0.95	0.89	0.92	3437
2	0.88	0.94	0.90	3158
accuracy			0.89	9824
macro avg	0.89	0.89	0.89	9824
weighted avg	0.90	0.89	0.89	9824

Table 4 | 3-Way Classification report

Precision and recall are performance measurements we mentioned during the course that helps us asses how well our model did. The F1-Score is the harmonic mean of them together and follows this equation:

$$F_1 = \frac{2}{\mathrm{recall}^{-1} + \mathrm{precision}^{-1}} =$$

$$= 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}} = \frac{\mathrm{tp}}{\mathrm{tp} + \frac{1}{2}(\mathrm{fp} + \mathrm{fn})}.$$

	precision	recall	f1-score	support
0	0.96	0.94	0.95	6553
1	0.89	0.92	0.90	3271
accuracy			0.93	9824
macro avg	0.92	0.93	0.93	9824
weighted avg	0.93	0.93	0.93	9824

Table 5 |2-Way Classification report

Error analysis

As already seen, the model that trained on the 2-way classification task achieved better results and supplied wrong classifications on only 651 (6.63% of the test set), while the 3-way classification model was wrong on 1040 <T', H'> pairs (10.586% of the test set).
We know that the data is labeled by 5 human labelers per sentence pair, and the "gold label" for each was set by the majority of the labels given. Therefore, we thought it may be interesting to see that the model made decisions that are closer to one's humans could have made:

We can see that the wrong predictions were made frequently when the consensus between the labelers was lower. Wrong predictions were made when the gold-label got an average of 3.7 (in both tasks), while correct predictions were made with average of 4.4 for the 3-way classification task and of 4.65 for the second task.

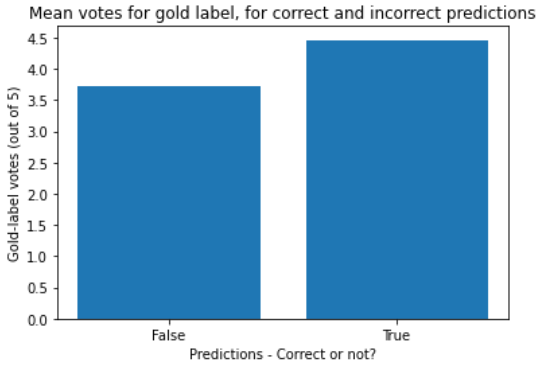

Figure 9 | gold-label vote count – 3-way classification

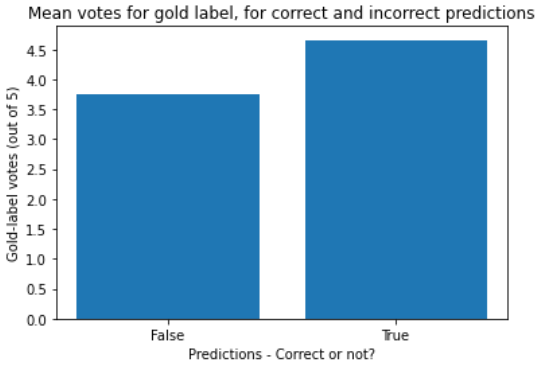

Figure 10 | gold-label vote count – 2-way classification

Quick look on the confusion matrices bellow shows that in both tasks most of the predictions were correct. In addition, we can learn that contradictions were easiest to find (yellow squares). The mix between neutral and entailment labels and predictions made the majority of wrong predictions for the 3-way classification task (almost 500!), whereas in the easier task, all the errors together are not far from this number. It seems like the addition of a "gray area" neutral label makes the model's task much harder and the ambiguity of the language is mostly expressed in those cases.

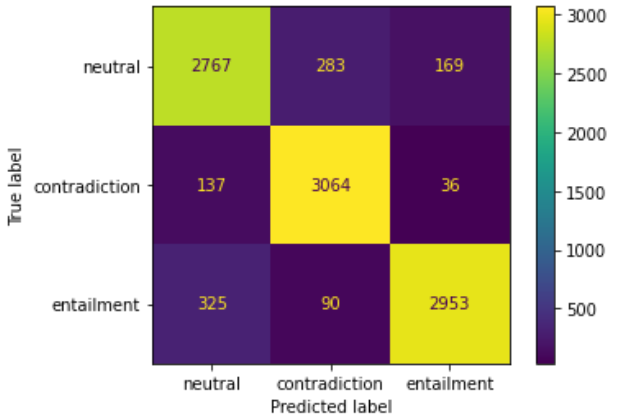

Figure 11 | Test Set Predictions; confusion matrix – 3-way classification

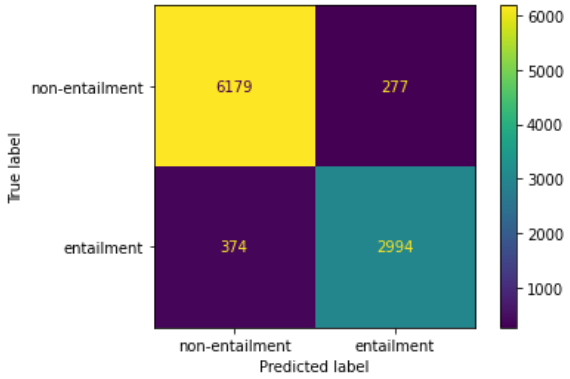

Figure 12 | Test Set Predictions; confusion matrix – 2-way classification

Figure 13 | Wrong predictions closer look

The chart to the left shows the failure count divided to cases. Two interesting things to point out here are the fact that the 2-way classification task failed by less than half of the other task, and the fact that the cases where both tasks failed to predict an entailment label makes more than half of the cases together.

Moreover, we thought that the length of the sentences may affect the accuracy of predictions. Quick look taught us that the lengths are close enough to make no real factor on the process: Sentences pairs on which both models predicted correctly included 13.9 and 7.45 tokens, while pairs with a double failure had 14.4 and 7.8 tokens per sentence.

Execution Instructions

- ⇒ The entire project is available in our GitHub repository.
- ⇒ To view every step's results, you can explore the JuPyter notebook here.
- ⇒ You may also run and play with the notebook in Kaggle.
- ⇒ You may also clone the repository and run the notebook.
- ⇒ Note that our re-trained BERT model and the encoding pickles are available for download from the repository's releases section, so you can download and use them instead of re-encoding and training. All you need to do is to download the data pickles & the models, and place them in `./data` and `./models` respectively.