Artificial Intelligence II
Assignment 4 Report

Andreas - Theologos Spanopoulos (sdi1700146@di.uoa.gr)

January 17, 2021

Exercises 1 & 2

For these exercises, publically available pre-trained models from the python library Sentence Transformer were used. A list with all the available models and their respective statistics, can be found here.

Preprocessing

From the dataset, the following fields are parsed and stored in the class "CovidArticle":

- ID
- Title
- Abstract Text, tokenized into sentences
- Body Text (section names and paragraphs), tokenized into sentences

For every piece of text, the following preprocessing pipeline is followed:

1. Remove URLs
2. Remove references to bibliography
3. Remove multiple full stops (e.g. ...)
4. Remove the "et al." string
5. Remove figure references

Numbers 1, 2 and 5 were removed because they offer no use in retrieving a sentence containing them. Numbers 3 and 4 were removed because they break sentence tokenization.

Every piece of text is broken down into sentences. A list containing all the sentences can be accessed in the CovidArticle.text getter method. This format is convinient because it allows the sentenced to be fed directly to the sentence transformers.

Sentence Embedding Approaches

The 2 models that were finally chosen, are:

1. stsb-distilbert-base. This model was chosen because of its good performance, and its high speed in computing the embeddings of sentences (4000 senteces/sec on V100 GPU).

2. stsb-roberta-base. This model was chosen as its the second best model regarsing the STSb performance, and its speed is acceptable (2300 sentences/sec on V100 GPU).

Both models are Sentence-BERT models, which basically means that they use BERT-like pre-trained models to compute the sentence embeddings.

Now let's discuss about how the models are used to retrieve relevant text for every query. The pipeline used is as follows:

1. Filter our articles that are irrelevant to the subject query.

2. From the articles kept, find the best one and return it.

Step 1

For every article, a "summary" is computed. This summary is the simplest possible: it's a list consisting of sentences which are:

1. The title of the article

2. The tokenized sentences of the abstract of the article

3. The sections of the article, each being treated as a standalone sentence.

Then for every model, we compute the sentence embeddings for each sentence in the summary. These embeddings are used for filtering our irrelevant articles: Articles where the highest cosine similarity with any sentence of the summary is lower than a specified threshold, are discarded. This helps us keep only relevant articles and therefore compute less sentence embeddings, as it is quite a costly operation, especially on low-end hardware.

Step 2

For the articles that have “survived”, compute the sentence embeddings for the whole text, and pick the one that has the highest cosine similarity with the query, on any sentence. Then, consider that sentence the passage. After that, start compuring the cosine similarity of adjacent sentences with the initial passage, and if they are above a 0.5 threshold, append them to the passage. This process repeats until either the threshold of 0.5 is not met on both sides, or we run out of the section in which the inital passage belonged.

Comparison

Some test queries have been hand-written for evaluation purposes. Those queries and the corresponding articles in which their answers lie, are listed in the queries.txt file.

In the task of finding the correct article containing the answer to a given query, the models performed:

Model / Query	stsb-distilbert-base	stsb-roberta-base
Query 1	✓	✓
Query 2	✓	✓
Query 3	✓	✓
Query 4	✓	✓
Query 5	✓	✓
Query 6	✓	✓
Query 7	✗	✓
Query 8	✓	✗
Query 9	✗	✗
Query 10	✓	✓
Total Accuracy %	80%	80%

Let's also take a look at the results in the task of returning also the correct passage along with the correct article:

Model / Query	stsb-distilbert-base	stsb-roberta-base
Query 1	✓	✓
Query 2	✓	✓
Query 3	✓	✓
Query 4	✓	✓
Query 5	✓	✓
Query 6	✓	✓
Query 7	✗	✓
Query 8	✓	✗
Query 9	✗	✗
Query 10	✓	✓
Total Accuracy %	80%	80%

The time required to compute the embeddings of all the summaries is

- 15 minutes for stsb-distilbert-base
- 27 minutes for stsb-roberta-base

The average time required to return the result of a query for every model is

- 17.75s for stsb-distilbert-base
- 24.91s for stsb-roberta-base

Note that these times get faster as more queries come in, because article text embeddings get saved in a dictionary.

Error Analysis

For an uncorrect article/passage to be returned, there are 2 possible errors that might have occurred:

- In Step 1, the correct article got filtered out.
- In Step 2, the correct article was kept but did not contain the sentence with the maximum cosine similarity.

Let's take a look in which case each of our query failures belong to.

By un-commenting the 2 print statements in the "find_best_article()" function, we can see which articles were found relevant. The others were filtered out. Let's make a table for every model to see what is going on:

stsb-distilbert-base

Failure Reason / Query Failures	Step 1 Error	Step 2 Error
Query 7		✓
Query 9	✓	
Total	1/2	1/2

stsb-roberta-base

Failure Reason / Query Failures	Step 1 Error	Step 2 Error
Query 8	✓	
Query 9	✓	
Total	2/2	0/2

From this small sample, we can deduce that the "summary" tecnhique does not work perfectly. This is expected, as the title + the abstract + the section titles do not capture the entirety of an article. For example, we may have an article about coronaviruses, but in the introduction there is a reference to the Spanish Flu, and the amount of casualties it caused. Since this information is irrelevant to the article itself and the pieces that make up the summary, it will be filtered out. Of course, there are way better methods to create summaries, like summarizers, etc. These haven't been tried, as I am running low on time. One solution could be to decrease the similarity threshold, so that more articles get included. This will slow down significantly the running time, so it is not an option. Therefore, improving the summary of each article should be the way to go for improving this text retrieval mechanism.

One other thing that could be done is to get rid of the summaries, and just pre-compute once the sentence embeddings for every article. On the Colab GPU, this procedure takes about 2 hours for a model like *stsb-roberta-base*. Since cosine similarity is not a very expensive operation, this would lead to the best results, as all the articles would be taken into account, but slower (on average) as it would compute $O(n)$ cosine similarities for every query, where n is the number of articles.

The above idea would technically give the most accurate results. Will it? There is a small catch. The problem lies on the selection of the "best" sentence. The sentence with the highest cosine similarity with the query is not always the answer to it. This problem becomes apparent when the embeddings have not been trained on specific domain-knowledge data, for example, text containing biological terms. The same problem would occur in the above models, had more queries been created. Thus, apart from the issue of identifying quickly the relevant articles, one needs to make sure that the remaining articles can be fully understood, that is, the sentence embeddings will capture all of its semantic meaning. This can be achieved by fine-tuning the models on the specific domain-knowledge data.

Conclusion

Taking into account the above results, model "stsb-roberta-base" can be considered as the best out of the two models, since

1. It filters out the most sentences (error analysis showed that it keeps around 5-25 sentences per article).

2. Even though slower, it finds the *exact* answer for most queries, unlike "stsb-distilbert-base" which sometimes finds *approximate* answers.

3. Once most sentence embeddings will have been computed, it will become faster than "stsb-distilbert-base", since it will only have to find the cosine similarities with many less articles.

Exercise 3

Even though there is an already build model for Question Answering from the Hugging Face library here, I re-implemented it myself using the vanilla BERT model. Same goes for the preprocessing of the SQuAD, I preprocessed it myself as it does not make sense to copy everything ☺.

Preprocessing

All the operations regarding the preprocessing of the SQuAD can be found in the squad_preprocessing.py file. The procedure can be described in the following steps:

1. Read the json file containig the SQuAD data. Parse it into a dataframe.
2. Preprocess the dataframe such each example gets translated to it's corresponding BERT input. This is achieved my tokenizing the question and the context, while also adding the [CLS] and [SEP] tokens. For some conrner case we have, we have the following rules:
    - Answers that contain only part of a token, are changed in order to contain the whole token.
    - BERT inputs that are longer than the maximum specified bert input length (hyperprameter set in the beggining), are shortened by removing the text that is the "most" far away from the answers. Note that this idea is stupid, as information about the target is "introduced" in the input. It was a simple solution to implement. A better idea would be to have a rolling window in the context, and if the original answer is inside that window, set it as the answer, if it is not, set the answer to "non-existing". This could also work as a way to augment the dataset. But since the examples where the input was longer than the allowed were few, I did not get to implementing this solution.
3. Pad the BERT inputs with 0 such that all the inputs have the same length, that of the input with the max length. Also add the Mask IDs and the segment IDs in the dataframe, in order to avoid doing it later.
4. Question that have no answer, are assigned to the class 0 ([CLS] token).

Model

The model is pretty simple. We run the input through the BERT, and get the embeddings of the last n hidden states. In the notebook implementation, $n = 4$ was used (which means that we stack the hidden states of the layers 9-12 of the BERT output). Then, the concatenated output is flattened, and fed into 2 linear layers (one for the start token and one for the end token), where their output shape is the length of the BERT input. Normally, we should use a softmax function on the output of the above layers to get the probabilities of each token being the start token and the end token respectively. We do not need to apply a softmax, as the CrossEntropyLoss from PyTorch does it. Some hyperprameters of the model are:

1. Learning Rate = 10^{-6}, for Adam
2. Regularization (weight decay) = 10^{-2} for the Loss Function
3. Batch Size = 16
4. epochs = 2
5. Max length of BERT input = 256

Performance

The metrics used to assess the performance of the model are:

1. Training Loss: 1.81, Validation Loss: 2.08
2. - Training EM: 49.223, (HasAns: 29.176, NoHasAns: 89.236)
   - Validation EM: 47.460, (HasAns: 7.726, NoHasAns: 87.081)
3. - Training F1: 64.965, (HasAns: 52.805, NoHasAns: 89.236)
   - Validation F1: 51.850, (HasAns: 16.518, NoHasAns: 87.081)

The model performed decently as we can see from the above scores. The graphs of the Losses during Training, computed as an average every ~ 500 batches of an epoch, can be found in the Notebook.

Reasons for the shortcomings of the model

The model is clearly overfitting on the training examples that have no answer. If we view this as a classification task (which technically it is), that is, we have maxlen (= 256) tokens (classes) for each example, we are trying to classify every token as “start token” or “end token”. Start and End tokens have many range of values (usually from 10 up to 255), while questions with no answer (which represent 1/3 of the dataset) have their start and end tokens values at 0. Hence, the overfitting. Ways to overcome this issue are:

1. Adding Dropout between the Flattening of the BERT output and the Fully Connected layer.

2. Increasing the weight decay factor (“amount” of Ridge regularization). Currently it sits at 0.01.

Of course, there are more reasons for why the model is not performing better, some of those are:

- Since there are severe GPU constraints, the maximum allowed length of the input was set to 256. BERT can actually handle inputs of length up to 512. The input was truncated to 256 by removing tokens far away from the actual answers. Therefore, some information loss may have occurred, that prevents the model from making actual good predictions. Exactly 12318 training examples have length bigger than 256, which is roughly 1/10 of the training dataset. I tried increasing the maxlen hyperprameter, but memory was always an issue, even with small batch sizes like 16.

- For the tokenization task, some questions where their answer is part of a token, but not the whole token, have “misleading” labels. This was not taken into account, as the model learns to predict answers only on the token level, and not on the character level.

Some ideas that could help improve the performance of the model and mitigate the above problems, are:

1. Increasing the maximum length of the BERT input to it’s allowed 512. For bigger inputs, use the “sliding window” technique mentioned in page 7.

2. Using bert-large instead of bert-base. The difference will be quite noticable, but training slower as well.

3. Using a scheduler for the learning rate (Reducing On Plateau should work).

4. Using a second BERT model, that would take as input the question and the answer tokens as a string of characters, and will learn to predict which characters are part of the actual answer, e.g. if the predicted answer it the word "sixth" and the ground trugh answer is "six", then we would feed to the second bert the question + "s i x t h", and make it predict the first 3 letters.

Comparison with fine-tuned model

The fine-tuned (on SQuAD) that was used for comparison is the "bert-large-uncased-whole-word-masking-finetuned-squad". On the SQuAD 2.0, it performed:

1.
   - Training EM: 50.476, (HasAns: 74.272, NoHasAns: 2.974)
   - Validation EM: 39.265, (HasAns: 75.641, NoHasAns: 2.994)
2.
   - Training F1: 58.286, (HasAns: 85.998, NoHasAns: 2.974)
   - Validation F1: 44.042, (HasAns: 85.209, NoHasAns: 2.994)

We notice that the model is not performing perfectly. This is due to the following reasons:

1. If I am not mistaken, the model was fine tuned on SQuAD 1.0, which did not contain questions with not answers. Hence, the low performance on questions with no answer (which in turns lowers the overall scores in EM and F1).

2. The tokenizer I used to do the tokenization was from the pre-trained model "bert-base-uncased", and not the one for the same pre-trained model, the "bert-large-uncased-whole-word-masking-finetuned-squad". Therefore, some mistakes could be happening in the tokenization, which affect the whole meaning of a sentence (or at least it tricks it), making it missclassify the start and end of a question.

3. From some question-answer examples, some tokens were removed from the context in order to comply with the max bert-input-length parameter with value 256. It is probable that some examples lose their “sense”, as some context being removed could make a question with answer, unanswerable,

Still, even though the model is not performing perfectly, it out-performed my model. This should not take you by surprise as

1. This model used bert-large instead of bert-base. The difference is quite noticable in any application. Bert-large may be slower as it has 24 attention layers instead of 12, but it’s embeddings are way more accurate.

2. I don’t have a GPU ☺. The fine-tuned model was probably trained on many GPUs, much better that the one Google Colab offers. I was not able to run a Grid Search or any other technique of hyperprameter searching to achieve better performance, as every epoch needs 1-2 hours.

With that said, the only advantage of my model compared to the fine-tuned, is speed. Since I am using bert-base, it is almost thrice as fast during inference.