Peyton Skwarczynski

CSI 4180 – Natural Language Processing

Dr. Steven Wilson

NLP Homework 3: Transformer-Based Models

1. Dataset

For this assignment, I chose to use the Huggingface SMS Spam Detection dataset. This public set is a collection of SMS messages that are either labeled as spam or legitimate messages (referred to as "ham"). Compared to some of the other datasets that could be used for this assignment, the SMS Spam Detection dataset is quite small, as it contains only 5,574 real English messages. Due to the size of the dataset, the developers did not split the data into different splits. Instead, they simply loaded a single training set. To split this into separate training and evaluation sets, I ended up taking an 80/20 split of the original dataset, where I then test the 80% training set on the 20% evaluation set. Additionally, it is important to note that the different SMS messages included range from strings as small as length 3, up to 911. This dataset was used universally throughout my project, for the fine-tuning of pre-trained models, zero-shot classification, and for creating BOW and other simple baselines. For each section of the assignment, the input is always the SMS message, with the output being whether or not the message is labeled as spam.

	Train Split (80%)	Test Split (20%)
Size of Data	4456	1114

2. Fine-tuned Models

The first pre-trained model that I used for fine-tuning was the prajjwal1/bert-tiny model. This model in particular is a Pytorch pre-trained model converted from the Tensorflow checkpoint found in the official BERT repository by Google. Due to computational limitations on my personal laptop, I resorted to the tiny version of this model, as this variation is the smallest BERT model available. The following chart displays the complexity of each of the BERT models, where L represents the number of stacked encoders, and H is the hidden layer size.

	H=128	H=256	H=512	H=768
L=2	2/128 (BERT-Tiny)	2/256	2/512	2/768
L=4	4/128	4/256 (BERT-Mini)	4/512 (BERT-Small)	4/768
L=6	6/128	6/256	6/512	6/768
L=8	8/128	8/256	8/512 (BERT-Medium)	8/768
L=10	10/128	10/256	10/512	10/768
L=12	12/128	12/256	12/512	12/768 (BERT-Base)

As for the size of this model, prajjwal1/bert-tiny contains approximately 14.5 million parameters, whereas the base version of BERT contains around 110 million. BERT was trained on the entire English Wikipedia and BooksCorpus, which contain 2.5 billion and 800 million words respectively. Additionally, the original BERT model was trained using 16 TPU pods over a time period of several days. We do not have access to the exact training specifications of the prajjwal1/bert-tiny model, but since it is simply a smaller version of the base BERT model, we can assume that it took significantly less time to train, since it uses a smaller amount of input data.

The second model I ended up using for fine-tuning was distilbert/distilroberta-base. This model acts as a case-sensitive distilled version of the RoBERTa base model, as it uses a DistilBERT training procedure. Compared to the bert-tiny model, distilroberta-base is significantly larger, as it contains 6 layers, 768 dimensions, and 12 heads, creating 82 million total parameters in the model. For comparison, distilroberta-base is twice as fast as RoBERTa base and contains around 40 million less parameters. Since this model is also a variation of the BERT base model, it was trained on the same data: the English Wikipedia and BooksCorpus. Also, similar to bert-tiny, the exact training specifications for distilroberta-base are not available to the public. For comparison, however, it should be noted that the original RoBERTa model was trained on 1024 32GB V100 GPUs for an entire day.

3. Zero-shot Classification

The first model that I used for the Zero-shot classification was mistralai/Mistral-7B-Instruct-v0.1, which is a transformer-based model developed by Mistral AI. This model is actually a pre-trained variant of GPT-3, but because of this, we do not know the exact training data of the model. One bit of information that has been published by Mistral AI is that mistralai/Mistral-7B-Instruct-v0.1 contains around 7 billion parameters, hence the “7B” in the name. With this taken into consideration, it is fair to assume that this model took multiple high-end GPU’s several days to complete, which is around average for models of this size.

The second model I ended up using was bigscience/bloomz-560m, which is a transformer-based model mainly used for language conversion. Compared to the 7 billion parameters of the previous model, bigscience/bloomz-560m only contains approximately 560 million parameters. By going on their Huggingface webpage, some of the training information, including the model, hardware, and software, are all readily available. This information can be seen in the screenshot below, which was taken directly from the Huggingface website.

Training

Model

- **Architecture:** Same as bloom-560m, also refer to the `config.json` file
- **Finetuning steps:** 1750
- **Finetuning tokens:** 3.67 billion
- **Finetuning layout:** 1x pipeline parallel, 1x tensor parallel, 1x data parallel
- **Precision:** float16

Hardware

- **CPUs:** AMD CPUs with 512GB memory per node
- **GPUs:** 64 A100 80GB GPUs with 8 GPUs per node (8 nodes) using NVLink 4 inter-gpu connects, 4 OmniPath links
- **Communication:** NCCL-communications network with a fully dedicated subnet

Software

- **Orchestration:** Megatron-DeepSpeed
- **Optimizer & parallelism:** DeepSpeed
- **Neural networks:** PyTorch (pytorch-1.11 w/ CUDA-11.5)
- **FP16 if applicable:** apex

After going through various lengths and styles of prompts for the zero-shot classification for both of my selected models, both showed difficultly understanding the dataset and, for the most part, gave inaccurate results. When prompting the datasets with simple templates like “Is this message spam? Answer ‘Yes’ or ‘No’”, the models would not initially answer with a single word, but instead a long string, or even multiple lines of text not entirely related to my initial question. This showed me to be more concise with my prompt and not leave room for potential open-ended answers. Even after extending my prompt with phrases like “Please answer the previous question using one word” or “Please answer the previous question using either ‘Yes’ or ‘No’ only” the models would still answer with random strings not pertaining to the original question. To make sure this issue was not with my models, I even tested the same prompts on a couple other models as well, with each of them resulting in a similar conclusion. Finally, I managed to come up with this prompt, "Is this text message spam? Answer 'Yes' if the message is spam, answer 'No' if you are unsure. Please only enter one word, either 'Yes' or 'No': ", and the models finally started to give me one-word “Yes” or “No” answers. The problem that I started seeing now was that no matter what I did to update or change the prompt, the models would either return a worse output that was not expected, as mentioned previously, or the model would return an extremely large amount of “Yes” values. This showed me that both of the models believed there to be a large quantity of spam messages, even though there was actually only 13% of spam messages in the entire dataset. Even though I changed and updated the prompt in many different ways, the models would always predict a large quantity of spam messages, leading to an extremely poor 15.2% accuracy.

Interestingly enough, this was the exact same accuracy percentage across the two completely different models I selected. Since the 13% of spam messages in the dataset are not equally distributed, and the fact that I am only using the first 1000 messages, the models seem to be responding "Yes" to every single message and then only getting correct the messages that are actually spam. An additional note I realized by looking deeper into the mistralai/Mistral-7B-Instruct-v0.1 model, is that input prompts are supposed to use certain tokens representing the beginning and end of sentences, [INST] and [/INST] respectively, but the results of the classification are valid even when those tokens are not included in the prompt, as shown by the 15.2% accuracy rating. When rerunning the prompt with the added tokens, the result yields an 84.8% accuracy, which is the exact opposite of the previous 15.2%. This seems to be due to the model completely changing its original result from labeling all messages as spam, to now labeling all messages as not spam.

My conclusion to this issue is that the models are easily confused when they come across difficult-to-recognize text. Since this dataset contains real text messages sent by the average, everyday individual, a majority of the messages include some sort of non-uniform communication, like slang and acronyms. Often times, these messages can only be deciphered by the individuals involved in the conversation, regardless of if the text is spam or not. Since the models are not used in chatbots, it would make sense that they have not been trained on anything outside of natural language. In fact, the Huggingface webpage for bigscience/bloomz-560m specifically recommends only using natural language as text input. Due to the training of these models, it is safe to assume that they are labeling all messages as spam or not spam in confusion and not being able to read the input message, rather than just being a poor model in general.

4. Baselines

For the baselines portion of the assignment, I ended up running 3 different baseline tests:

The first baseline that I ran was a Multinomial Naïve Bayes Baseline. Referencing some of the code used in previous assignments, I converted the different splits of the dataset into a preprocessed Bag of Words (BOW) format. Next, I needed to convert each of these two splits into a matrix format that is suitable for the Naïve Bayes classifier. Then, using the training data, I passed the test split through the Multinomial Naïve Bayes classifier to estimate the results of the test set, based on the actual results of the training split. I also used a simple accuracy calculation to show the probability that the Multinomial Naïve Bayes classifier was correct in predicting SMS spam messages.

My second baseline was a simple random baseline, which operated by randomly assigning a 0 or 1 to each SMS message in the test data split. If the 0 was assigned to the SMS message, then it would be labeled as a legitimate message. If a 1 was assigned to the message, then it would be qualified as a spam message. The accuracy of this particular baseline was calculated by comparing the random assignment of 0's and 1's to the actual labels assigned for each SMS message in the dataset.

The third and final baseline was a Target-Class baseline, which works by assigning the target-class to each SMS message in the test data split. For this particular dataset, the target class is 1, or the label of a spam message, because the most important factor is if the particular SMS message

is, or is not, a spam message. The accuracy of this baseline was then calculated by comparing the assignment of the target-class with the actual label for each SMS message in the dataset.

5. Results

Type:	Model:	Accuracy:
Fine-Tuned Model	prajjwal1/bert-tiny	0.9919210053859964
Fine-Tuned Model	distilroberta-base	1.0
Zero-Shot Classification	mistralai/Mistral-7B-Instruct-v0.1	0.152 OR 8.48
Zero-Shot Classification	bigscience/bloomz-560m	0.152
BOW Multinomial Naïve Bayes Classifier	N/A (Dataset Only)	0.9802513464991023
Random Baseline	N/A (Dataset Only)	0.49371633752244165
Target-Class Baseline	N/A (Dataset Only)	0.13016157989228008

When looking at the results for both of the Fine-tuned models, we can see that these contained the highest accuracy out of all classification methods. I believe that this is because the dataset is working alongside a previously trained model to make such classifications. By running the already accurate model through pretraining on the particular dataset, you are able to create a highly accurate, newly trained model great for classification. On the other hand, these results are almost the complete opposite result of the Zero-shot models. Since the models for Zero-shot classification are never ran through dataset pretraining, the models must have already been trained on a similar dataset, or else there will be extremely low accuracy ratings. As previously discussed, since the models I selected for the Zero-shot did not contain training on datasets related to natural communication, as through texting, the models became confused when looking at the SMS messages in the dataset, resulting in poor classification accuracy.

As for the results of the BOW Multinomial Naïve Bayes Classifier, this baseline proved to hold extremely high accuracy. This high probability is expected because rather than using a model to make predictions on the dataset, the dataset itself is being used to make predictions on a portion of its own data. Similar to the Naïve Bayes Classifier, the Random Baseline also has a completely expected accuracy, as it is close to 50%. Since there are only two possible labels in this dataset, it would make sense that the random baseline accuracy is very close to 50%, which it is. On the other hand, the Target-Class baseline accuracy is quite unexpected. For some reason, this baseline showed extremely low accuracy. This is due to the fact that the overall dataset only contains approximately 13% spam SMS messages, despite spam being the target class. It would make sense that the accuracy for this baseline is low because most of the actual labels in the dataset are not spam, while the configuration of the baseline sets the expected label as spam.

Overall, I would say that fine-tuning models to a dataset are the best way to generate accurate and consistent classification models. The added element of model fine-tuning ended up increasing the overall classification accuracy to near perfection, no matter what pre-trained model was used. When working with extremely large datasets, unlike the dataset shown here, there is an increased need for accuracy when working with classification models. When parameters are increased to billions, even a small percentage subtracted from the accuracy rate can potentially mean that

millions of classifications are being identified incorrectly. In a professional environment, this is not exactly what you would like to occur, hence the need for higher classification accuracy.

6. Reflection

Some of the things I learned throughout the completion of this assignment were improved research skills and problem-solving techniques. Section 2.1 and 2.4 of the assignment were not difficult for me to figure out, but significant challenges arose when trying to complete sections 2.2 and 2.3. When using Huggingface datasets and models for computation, the code, under normal circumstances, automatically transfers the high processing load over to the device's GPU for parallelization. When working on this assignment, however, I very quickly realized that this was not the case for my current setup. For some reason, when trying to execute and test the code, I continually received runtime estimates as 5+ hours. From using online resources, I saw that the fine-tuning of models should on average take around 5 minutes, depending on the size of the dataset and complexity of the model. This being said, my code presenting an estimated runtime of 5 hours was extremely surprising. After further investigation, I realized that I could get the models and datasets sent over and stored in my GPU's memory, but the processes would only ever be executed on the CPU. This actually caused greater estimated execution time than trying to run the code strictly on CPU, due to increased overhead of communication between the CPU and GPU.

After reaching out to the Professor, the teaching assistant, and fellow classmates, I found myself with no clear solution for my configuration. After hours of research and attempts at changing the models and datasets, I tried copying my setup over to a Mac laptop, as I was previously using Windows, and the code was able to run successfully with runtime of around 5 minute. From this I could confirm that the code itself was not the issue, but rather the Windows environment I was using to run the code. By comparing the process of package installation across the two operating systems, there was absolutely no difference. The only two differences between the configurations were that:

1. When installing Torch using the command line, Windows and Mac OS both install the latest version, but on Windows, the installed package is labeled as either "version number +cpu" or "version number +gpu". The version of the package on Mac does not have any string concatenated onto the end of it.
2. Since Mac laptops do not have NVIDIA GPUs, but rather their own Mac GPUs, there was no need for installing CUDA on the Mac laptop.

I expect this environment problem to be caused by one, if not both, of these observations. Due to time constraints with trying to get the rest of the assignment done as quickly as possible, I currently have not had the chance to go back and figure out a solution. Instead, I completed the rest of the assignment on the Mac laptop and a separate Google Colab notebook.