Report

Dataset

The task that I was working on was **Hate Speech Detection** i.e. classifying texts as **'hatespeech'**, **'not-hatespeech'** or **'neutral'**. For context, the definition of **Hate Speech** is as follows:- *abusive or threatening speech or writing that expresses prejudice on the basis of ethnicity, religion, sexual orientation, or similar grounds*. I used the `ucberkeley-dlab/measuring-hate-speech` dataset from the **hugging face** link that was provided to us by the professor.

The dataset was collected by UCBerkeley in the form of **50,000 social media comments** sourced from **YouTube, Twitter, and Reddit** and labeled by 10,000 United States-based Amazon Mechanical Turk workers to measure a continuous spectrum from hate speech to counterspeech.

The **inputs** for this task are **texts** of social media comments which is a string and the **outputs** are class labels indicating the level of Hate Speech. If the **label is 0**, it indicates that the text is **non-hateful**. If the **label is 1**, it signifies that the text is **neutral**. If the **label is 2**, it denotes that the text is **hateful**. I am using **accuracy** as the evaluation metric for this task.

The following table showcases specific information about the data splits:-

Split	Size	Not-Hateful (0)	Hateful (1)	Hateful (2)
Train	108444	64601	7094	36749
Test	27112	16023	1817	9272

In simple terms the split was 80% training and 20% test dataset.

Other Dataset Features:

Average text length (words) - Train: 27.01, Test: 27.02
Languages: Primarily English (based on dataset documentation and inspection)

Fine-tuned models

The models I selected were the `google-bert/bert-base-uncased` and `distilbert/distilbert-base-uncased`. I selected these models by searching **BERT** on **Hugging Face** and setting the filter to **"text classification"** in the **Natural Language Processing** category.

The `google-bert/bert-base-uncased` is a **BERT** model pretrained on a large corpus of English data in a self-supervised fashion. This is the **"base"** version of the model which is a relatively moderate size as compared to some of its larger counterparts. I chose this size specifically because it can be trained in a reasonable amount of time. This model is **"uncased"** i.e. doesn't distinguish between capital lettered and lowercased words.

The `distilbert/distilbert-base-uncased` is the **distilled** version of the original **BERT** based model. It was also pretrained on the same corpus in a self-supervised fashion, using the BERT base model as a teacher. This means it was pretrained on the raw texts only, with no humans labelling them in any way (which is why it can use lots of publicly available data) with an automatic process to generate inputs and labels from those texts using the BERT base model. This is also the **"base"** and **"uncased"** version of the model.

In order to **fine-tune both models**, I followed all the steps given in the hugging face fine-tuning models guide which was shared by the professor as part of the assignment. I used `AutoModelForSequenceClassification` and `AutoTokenizer` from Hugging Face to load both `google-bert/bert-base-uncased` and `distilbert/distilbert-base-uncased` (in different files). Then I tokenized the text using the BERT tokenizer, convert float labels to int and formatted the dataset for **PyTorch**. Then I used `TrainingArguments` to configure **hyperparameters** and the `Trainer` API to fine-tune the model on the training split and evaluated on the test split. As previously mentioned, the metric I used to evaluate my dataset was accuracy, which I computed using the `evaluate` library.

The size of `google-bert/bert-base-uncased` is **110M parameters**. The model was trained on 4 cloud TPUs in Pod configuration (16 TPU chips total) for one million steps with a batch size of 256 and was trained for 4 days. All of this information is mentioned in their respective hugging face documentation. I fined-tuned this model using the available GPUs provided McMaster CAS department. Training this model took me about ~38 minutes.

The size of `distilbert/distilbert-base-uncased` is **67M parameters**. The model was trained on 8 16 GB V100 for 90 hours. All of this information is mentioned in their respective hugging face documentation. I fined-tuned this model using the available GPUs provided McMaster CAS department. Training this model took me about ~27 minutes.

Zero-shot classification

For the zero-classification task, I used the fourth method suggested by the professor which was to use a **Natural Language Inference**. I used two pretrained NLI models to classify the texts. As mentioned and the core task in this part of the assignment, I didn't pretrain the model. Instead I fed it all the texts in the test set with an adequate prompt that I found useful for the task.

The NLIs I used were `facebook/bart-large-mnli` and `MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli`. I chose `facebook/bart-large-mnli` as the first model because it was used in the reference tutorial provided by the professor, which mentions NLIs as the fourth method for zero-shot classification. I was browsing Hugging Face's model hub and applied the filter for '**Zero-Shot Classification**' in the '**Natural Language Processing**' category. I then sorted the results by 'Most Downloads,' and the top result that appeared was `MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli`. This is why I decided to use this particular model.

The `facebook/bart-large-mnli` is a **BART** model originally designed for sequence-to-sequence task. I used it the **"large"** variant of this model and its size is **407M parameters**.

The `MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli` is a **DeBERTa** model, which is an advanced version of BERT model due to its improved attention mechanisms. The size of this model is **184M parameters**.

The `facebook/bart-large-mnli` was trained on the `nyu-mll/multi_nli` which is a multi-genre Natural Language Inference (MultiNLI) corpus is a crowd-sourced collection of 433k sentence pairs annotated with textual entailment information.

The `MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli` was trained on the `fever/fever`, `nyu-mll/multi_nli` and `facebook/anli` datasets, which comprise 763 913 NLI hypothesis-premise pairs. All of this information is mentioned in their respective hugging face documentation.

I had some difficulties in understanding this task initially because I was confused as to what exactly we're supposed to prompt about. Initially I used prompts like `"Is this text hateful: yes or no: {text_from_the_test_dataset}"`, `"Classify this text as hateful or not: {text_from_the_test_dataset}"`. After using both these prompts, I remembered the professor's advice of "looking at the data" and redesigned the prompts. The dataset has 3 labels for hatespeech classification: 0 → not-hateful, 1 → hateful and 2 → hateful. I then created the prompt as `Classify the text as hateful, not-hateful or neutral: {text_from_the_test_dataset}` with the labels to classify being **"hateful"**, **"not-hateful"** and **"neutral"**. I didn't notice a substantial difference in the accuracy after changing the prompts. In my opinion, the reason could be that in NLIs, I already give it the labels it's supposed to predict, so regardless of whatever prompt we prefix the text by, the NLI gives predictions based on how "close" it is to those categories. Prompt engineering would be very useful in cases where people are using different methods like APIs (openAI, google gemini, etc.) where the model simply outputs "text" for an input prompt. In such cases we have to manually engineer the output text to be the exact "label" that we want, so that we can simply count them and then calculate the accuracy.

I learnt how to use NLIs using the reference tutorial which was given by the professor in the assignment. I tweaked certain parts to match the ground labels from the dataset and reused parts of code from task 1 to evaluate accuracy.

The zero-shot model was run on the GPUs provided by the McMaster CAS department. This process took me ~1 hr 25 mins because my code was a bit inefficient since it was computing everything sequentially.

Baselines

I computed three baselines to compare against my fine-tuned BERT and zero-shot models for the task. I used **logistic regression** to calculate the main baseline. Additionally, I calculated majority and random baselines as "**sanity checks**".

For the **BoW classifier**, the dataset was `ucberkeley-dlab/measuring-hate-speech` which is obviously the same as our fine tuned models because we're going to compare this with them. I split the data with the training data being 80% and the test data being 20%. I used the `TfidfVectorizer` from the **sklearn** library to convert the input features into **TF-IDF** form by transforming the data to fit the vectorizer. In order to improve efficiency, I limited the vocabulary to **5000** most frequent words. I then trained this for the training set as a **multiclassifier** for a **1000** iterations. After getting the predictions for the test set by using the trained classifier, I computed its accuracy using the `evaluate` library.

For the **majority classifier**, I first calculated the **frequency** of each class (0, 1 and 2) using `NumPy` in order to identify the **most frequently occurring** class. After identifying the most frequently occurring class, just create an arbitrary predictions array which is generated by assigning the value of the majority class to all the examples in the test set(done by just multiplying the size of the test set to a singular array with the value of the majority class label). We then **evaluated** the accuracy of the model by comparing the majority class predictions with the actual ground labels.

For the random baseline classifier, I just used `NumPy` to **randomly** assign any of the three classes to each test example. Then I simply compare these randomly generated labels to the actual ground labels in order to **evaluate** the accuracy.

The train and test sets were split with a seed of 42. Additionally the random baseline also used the same seed to maintain consistency.

Results

The following table showcases all of the results for the respective models:-

Model	Accuracy
Fine-tuned `bert-base-uncased`	78.87%
Fine-tuned `distilbert-base-uncased`	78.89%
Zero-shot `bart-large-mnli`	34.2%
Zero-shot `DeBERTa-v3-base-mnli-fever-anli`	34.2%
BoW baseline using Logistic Regression	78.3%
Majority baseline	59.1%
Random baseline	33.2%

The fine-tuned models `bert-base-uncased` and `distilbert-base-uncased` achieved the highest accuracy. The interesting thing to note here is that despite having less parameters, the `distilbert-base-uncased` was almost as accurate as `bert-base-uncased` which had substantially larger number of parameters. In my opinion, this is likely due to the fact that `distilbert-base-uncased` has a distilled architecture (as the name suggests) which preserves the key capabilities of the original BERT model while being more efficient.

In contrast, the zero-shot models performed poorly with both of the models achieving a 34.2% accuracy. This performance is barely above the random baseline accuracy which in my opinion showcases the edge fine-tuned models have over out-of-the-box **pretrained** models. This just serves as some kind of a lower bound because they're barely better than random guessing.

Overall, the fine-tuned models outperform the majority and random baselines, but I feel the results aren't impressive due to the fact that there is an extremely small difference between BoW baseline and the fine tuned model. I feel that the additional complexity and power of the transformer models may not be necessary for this task as simple models are already highly effective.

Therefore, after considering these results, if I were to give recommendations to someone else who was trying to build a model for this task, the first thing I'd suggest is to avoid zero-shot classification because in my experience they had a poor accuracy (34.2%) and took the most time to run (~1hr 25 mins). This poor performance doesn't even remotely justify the computational costs of running these zero-shot models. I'd recommend fine-tuning a lightweight transformer like `distilbert-base-uncased` (because it was almost as accurate as `bert-base-uncased` while requiring substantially less resources to run) which achieved 78.89% accuracy, balancing performance and efficiency. If resources are limited, then to be honest a simple BoW models is also highly effective, but the reason I recommend using a transformer over a simple model if you have the resources is that I didn't perform extensive hyperparameter tuning or rigorous experimentation with the models because this was beyond the scope of this assignment, however, this might not be the case for someone who's just trying to build an optimized model. In short, transformers always have a higher scope of outperforming simple models, so I don't see a reason to not use them when someone has the computational resources.

Reflection

The key learning from this assignment for me was understanding the trade-offs of using various kinds of models for a given task based on the requirements. Prior to doing this assignment, I had this biased notion that if anybody wants to make a good model, they just need computational resources because a computation powerful model will always outperform a simple model. These biases were instantly broken when I saw that zero-shot models, which, according to me, required the most computational power (because I used the "large" variant for bart mnli), achieved an accuracy far worse than its BoW counterpart, which wasn't even nearly as computationally heavy as the zero-shot models.

I was excited to check the outcomes given by the zero-shot models on trying different prompts, but I was actually surprised to see that the outcome was more or less the same. I asked my peers who used NLIs while attempting this assignment and they had the same experience as well. Additionally, one thing that confused me was the accuracy of both the zero-shot models. To my surprise, they had the exact same accuracy. While implementing the code to calculate the accuracy, I exactly followed the tutorial suggested by the professor and reused some part of code from task 1. In my opinion I don't think the code is faulty, I feel due to the fact that the one of the datasets on which the DeBERTa mnli was trained on is the same as the BART mnli dataset, the models had identical predictions.

Some other things I learned while doing this assignment were that students had access to expensive GPUs provided by McMaster and fine-tuning a random model for a specific task through the extensively documented tutorial provided by the professor. Additionally, I have used ChatGPT to look up certain syntax when I used scikit-learn to create the BoW classifier and to write the README.md