DATA471 Final Report

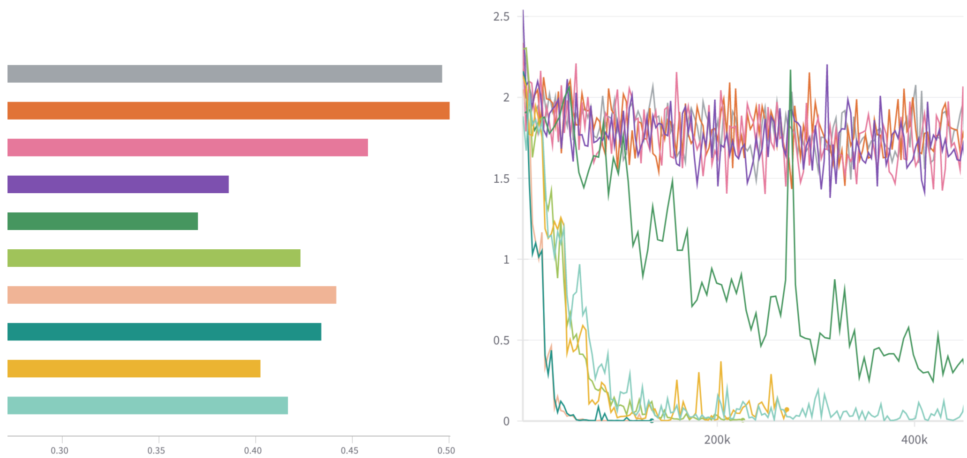
Task 1: Term-Frequency Vectors
Lead: Maxwell Schultz

Methods:
During the initial phase of the project I simply explored the libraries and methods available for implementing ML models in python. I ended up deciding that implementing a NN in pytorch could be both versatile in how I used it given that it was a categorization problem, and offer many hyperparameters with which I could use to tune the model. In the beginning I implemented a single-hidden-layer network and did some initial testing to get a benchmark on initial stochastic performance, this initial benchmark hovered around (35%) accuracy on the dev set. After this I implemented a multi-hidden-layer network, with which I could compare performance and get access to yet another tunable hyperparameter.

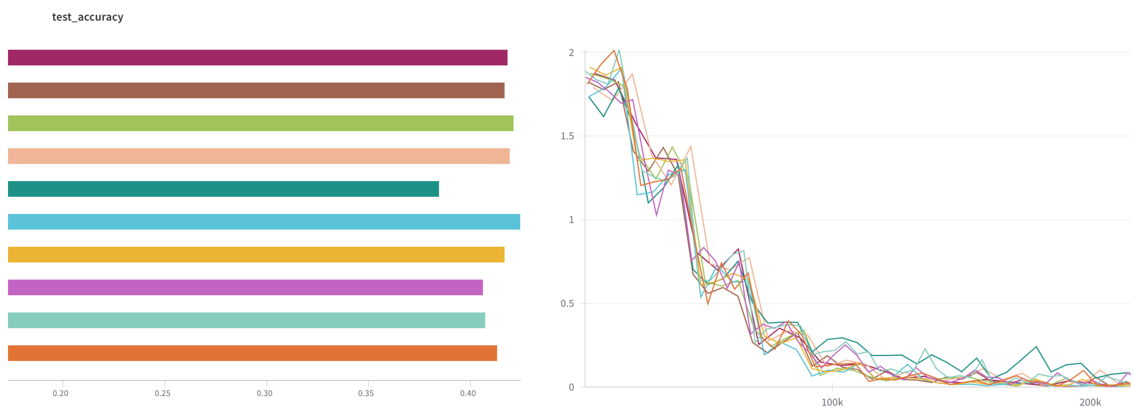
The next stage was actually working with the data and seeing which parameters I could change without the aid of a hyperparameter tuner, which yielded basic intuitions about what I should train and focus on. The initial thing I focused on was the multi-layer network, and how it fared in stochastic scenarios against the single-hidden-layer one. What I quickly learned was that extra layers did not appear to be helpful in this problem, and without very particular tuning, yielded worse results than the single-layer model. With this knowledge, I used weights and biases to begin doing basic recorded tests on my single-layer model. Using wandb, I began doing a suite of tests using the ReLU function, and switching to LeakyReLU, and Sigmoid to see if they had any useful effect. The activation functions did not have a measurable effect on the accuracy of the model.

The next stage was using wandb to build a parameter tuner, which allowed me to focus on which hyper parameters would be used for my best run. I initially used the random mode to save resources and time but also wanted to see the effect of the bayes mode on my model. Out of the various random iterations that I ran, the best accuracy I was able to boost the model to was approximately (50%), which I felt was a marked improvement from the initial (35%) baseline. Using the bayes version, it built on initial estimates of approximately (38%), and worked its way to a plateau of around (42%). Using my best runs from the various benchmarks and tests, I found that batch_size, learning_rate, and optimizer were the best predictors of a good run. I set a weighted average of the values from the best three runs, and used that to train the model on the entire dataset.

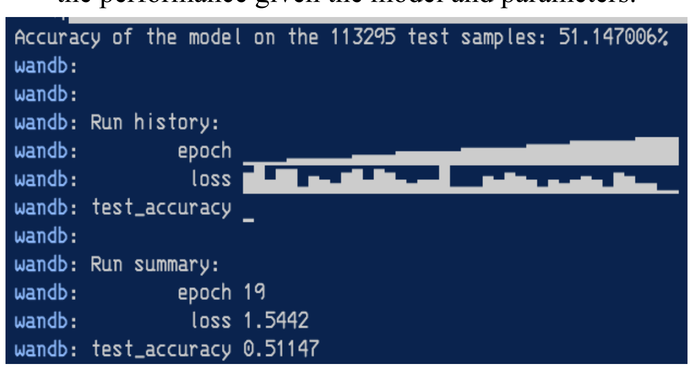
Model Details:
The model is a forward-feeding perceptron implemented using the pytorch nn.Module, with an input layer of size 100000, a hidden layer of size 15, and an output layer of size 25. The output of the first layer is passed through the ReLU function and the output layer is sent through the nn.CrossEntropyLoss function which is used for loss. The optimizer is the torch.optim.SGD optimizer. The model is trained over 20 epochs, with a batch_size of 140, and a learning rate of 0.085. The sparseX files are parsed, sliced into separate indices and values tensors, put through the torch.sparse_coo_tensor, of which the output calls its own to_dense() function to get the training data. The data is normalized with the F.normalize function, with the norm exponent being a p of 2.



A variety of tests on the single-layer model without a hyper-parameter tuner. This shows initial benchmarks for accuracy (left) hovering around 35-38% accuracy. The loss (right) can be seen as some of the weaker models having deep loss curves, while the stronger models quickly found good gradients and plateaued early.



The main hyper parameter tuning test results, accuracy (left) maintained an avg of around (40%), the loss function (right), decreased along all of the runs in roughly tandem. The tuner hit a plateau for increasing the performance given the model and parameters.



```
Accuracy of the model on the 113295 test samples: 51.147006%
wandb:
wandb:
wandb: Run history:
wandb:          epoch
wandb:           loss
wandb:  test_accuracy
wandb:
wandb: Run summary:
wandb:          epoch 19
wandb:           loss 1.5442
wandb:  test_accuracy 0.51147
```

This is the output of my best model, which has an accuracy of roughly 50.15%. The loss function is relatively flat throughout the run, which is expected given the previous results. I interpret this as a sign that the model found effective weights and biases quickly and efficiently, with extra epochs not adding much proformance. As a roughly 10-15% increase from my baseline, I consider this to be a very nice result considering the limitations of the model.