Task 4 Report

Io Paul
DATA 471
06/05/2025

1 Methods

We began working on this task by using the SVC package from the scikit-learn toolkit to implement support vector based classification models. After getting very poor results, we looked through the documentation for ways to tune the model better and found that the LinearSVC package is recommended for large, dense datasets. We began using LinearSVC models and found that the results improved, as well as the runtime. There are also fewer parameters, which allowed us to focus more on tuning certain meaningful parameters. We focused on tuning the C parameter, which represented the intrusions that the margins would allow. This had little impact on the accuracy of the models, so we focused on other parameters such as dual and found that they also had little impact on the accuracy scores. This seemed to indicate underfitting, so it seemed like the model needed to be more expressive.

In order to increase expressivity, we tried implementing a neural network model. The model outperformed LinearSVC models by a decent amount (about 10%). We tuned the model and still found that it was stagnating somewhat, and rather than continuing with hyperparameter tuning for a neural network, we decided to try another model type. This final model type was XGBoost, because we thought that an ensembling method would perform well on a classification task like this which had many different classes.

Our XGBoost models proved quite successful, as it took only minor tweaks to hit the scores that the LinearSVC models seemed stuck at. One method we tried with both LinearSVC models and XGBClassifiers was Z-score normalization, which we applied to the features of both the training and dev sets during pre-processing. This had almost no impact on accuracy, and ultimately was discarded. We continued tuning the XGBClassifier models, adjusting the parameters slightly to see how each works with the data. For example, using more estimators significantly increased accuracy. Another adjustment that seemed to work well was using a larger learning rate. This was surprising at first, as we initially assumed it would perform better with a lower learning rate since this was the case when building the neural network. However, we found that the best performing learning rate was 0.1. Anything larger than that led to poor results, and anything smaller than 0.1 resulted in almost no change in accuracy. After reading through the documentation, we realized that this was because the learning rate for XGBoost is completely different from the learning rate in neural networks, and is more related to how the trees are corrected. Increasing max depth also seemed to increase accuracy, but we soon found that anything above a depth of 10 led to overfitting. We believe this is because the depth parameter increases variance, so we kept the max depth low (at a value of 5 in the final model) to reduce variance.

The first baseline we considered was always predicting the majority class, since that was the lowest baseline for multiclass classification. This would give an accuracy of 12.65% for the training set and 14.22% for the dev set, which we found by calculating the relative percentages of each class in the total dataset. This calculation was performed by a small script that counted how many times each class appears in the target files, then divided that by lines in the file. This followed the principles of the accuracy formula we discussed in class, which is $\frac{TP+TN}{TP+TN+FP+FN}$.

2 Submission Model Details

The best model that we found for this task was an XGBoost XGBClassifier model implemented using SciKit-Learn library as well as NumPy and XGBoost. There was no preprocessing used, as it resulted in minimal change in accuracy. The hyperparameters chosen included the default parameters for the

model type, with the following exceptions: n_estimators = 1000, max_depth = 5, learning_rate - 0.1, and objective = 'multi:softmax'.

3 Results

3.1 Support Vector Machine (LinearSVC)

When using LinearSVC models, our accuracy scores were consistently hitting 35% for the training set and 36% for the dev set. The 7 models testing different hyperparameter configurations differed by at most 0.1%. This means that the results were about twice as good as the baseline, but still stagnant.

3.2 Neural Networks

The neural network models we tried had a higher accuracy than the LinearSVC models, reaching an accuracy score of 42.45% for the training set and 43.25% for the dev set. This was one of the highest performances, so it seemed like a good final model. However, tuning it was less effective than tuning the XGBoost models, and the XGBoost models actually outperformed the neural network slightly for this particular task given the tuning that they underwent.

Model	learning_rate_init	learning_rate	Train Accuracy	Dev Accuracy
nn_model_6	0.1	constant	42.45	43.25
nn_model_15	0.01	constant	36.03	36.85
nn_model_16	0.1	adaptive	42.5	43.13
BASELINE			12.65	14.22

Figure 1: Neural network MLPClassifier model parameter configurations and accuracies

3.3 XGBoost (XGBClassifier)

XGBClassifier models had the widest range of accuracies, though they still stayed relatively consistent. Figure 2 depicts the accuracies of the different models and highlights major changes in the parameter configurations. The XGBoost models were slightly more prone to overfitting than the neural network models (see xg_model_11 for an example of extreme overfitting), but became more stable once we figured out that the max depth parameter was causing this to occur. Because we could tune that parameter to avoid overfitting, and because the XGBoost model family consistently performed as well as the neural network models for our task, it seemed like a good candidate for the final model.

Model	Estimators	Depth	Learning Rate	Train Accuracy	Dev Accuracy
xg_model_9	2	2	0.1	31.12	32.07
xg_model_10	5	4	0.01	35.14	36.07
xg_model_11	20	20	0.01	67.21	42.5
xg_model_12	10	10	0.1	47.45	41.74
xg_model_13	500	5	0.1	41.36	41.4
xg_model_14	1000	5	0.1	53.5	44.07
BASELINE				12.65	14.22

Figure 2: XGBoost XGBClassifier model parameter configurations and accuracies

4 Distribution of Work

During this project, Jenny Sims helped in the data loading process by suggesting the numpy.loadtxt() method, rather than trying to transfer everything into a pandas DataFrame with little success. Additionally, her success with the XGBoost model contributed to the decision to utilize the same model for this task. Andy Ngo contributed to the organization of our repository and our reports, standardizing the titles and author formats as well as helping with organizational and formatting questions. Josh Crain assisted with the fundamental understanding of sparse and dense files, as well as with how to parse the datafiles.