Assignment #2, Drug Activity Prediction

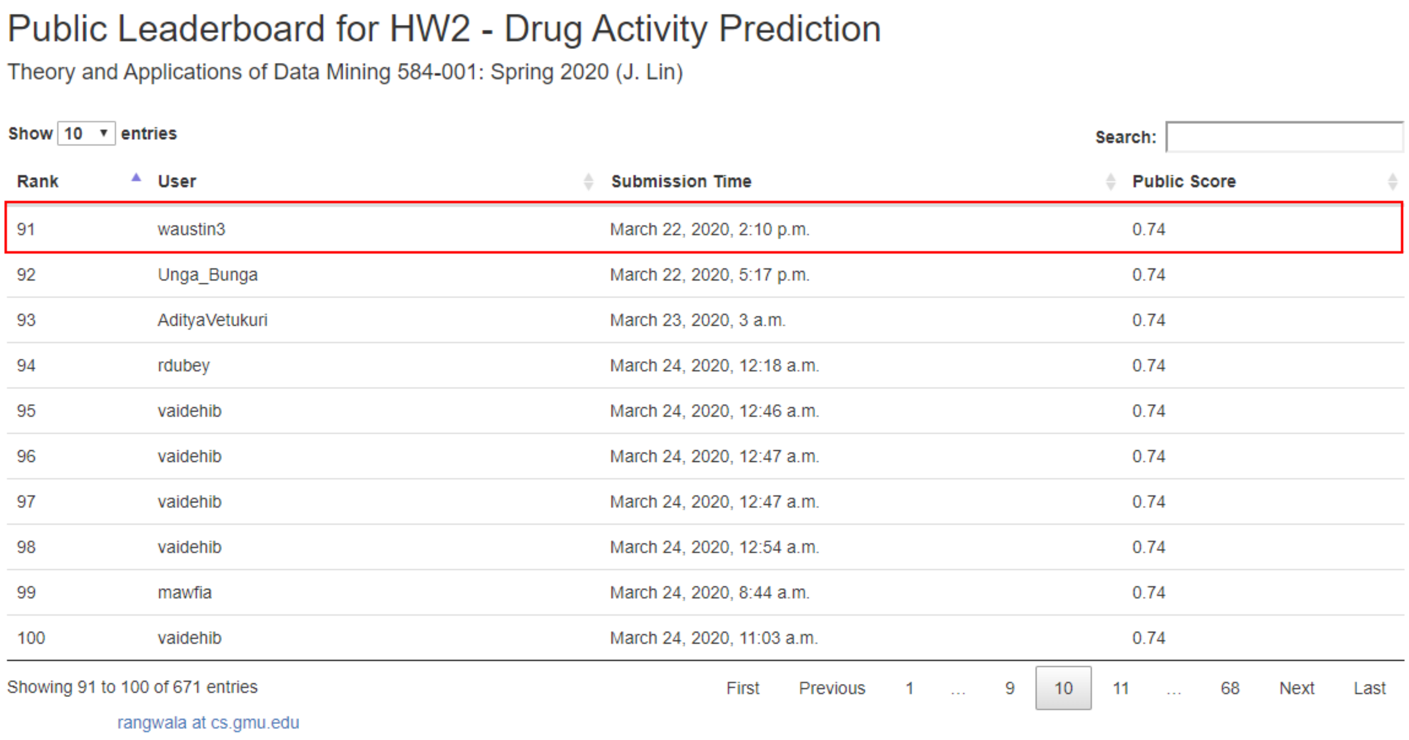
CS 584: Data Mining
Due March 24th, 2020
Submitted by William Austin

1. Miner Website Credentials

Registered Miner Username: **waustin3**

2. Rank & Accuracy of Submission

My final (and best) Miner submission for this assignment, had an accuracy of **0.74**, which ranks 21st in the class (not including multiple submissions).


Public Leaderboard for HW2 - Drug Activity Prediction
Theory and Applications of Data Mining 584-001: Spring 2020 (J. Lin)

Show 10 entries

Search:

Rank	User	Submission Time	Public Score
91	waustin3	March 22, 2020, 2:10 p.m.	0.74
92	Unga_Bunga	March 22, 2020, 5:17 p.m.	0.74
93	AdityaVetukuri	March 23, 2020, 3 a.m.	0.74
94	rdubey	March 24, 2020, 12:18 a.m.	0.74
95	vaidehib	March 24, 2020, 12:46 a.m.	0.74
96	vaidehib	March 24, 2020, 12:47 a.m.	0.74
97	vaidehib	March 24, 2020, 12:47 a.m.	0.74
98	vaidehib	March 24, 2020, 12:54 a.m.	0.74
99	mawfia	March 24, 2020, 8:44 a.m.	0.74
100	vaidehib	March 24, 2020, 11:03 a.m.	0.74

Showing 91 to 100 of 671 entries

First Previous 1 … 9 10 11 … 68 Next Last

rangwala at cs.gmu.edu

3. Instructions for Running the Code

For this project I chose to use Python because of the library support for creaing different models and performing other common machine learning tasks.

My python environment is based on **Conda 4.8.3**, which uses **Python 3.7.4**. The main libraries that I used for the project are:

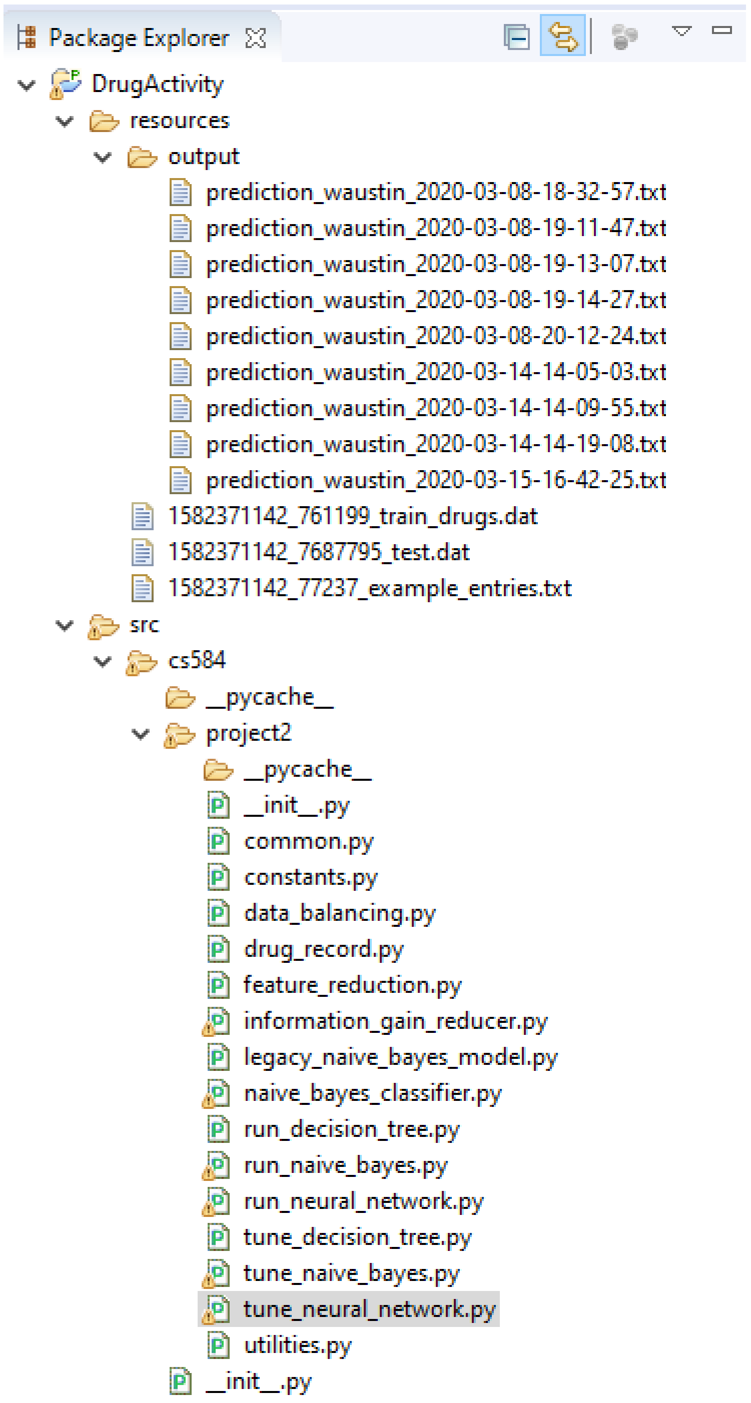
Library	Version	Website
numpy	1.16.5	https://numpy.org/
scikit-learn	0.21.3	https://scikit-learn.org/stable/
imbalanced-learn	0.5.0	https://imbalanced-learn.org/stable/index.html

Note that imbalanced-learn does not come with conda by default, but it can be installed via:

```
conda install -c conda-forge imbalanced-learn
```

This is the only change that I made to the default conda environment for this project.

For development, I am using Eclipse with the PyDev module. The project setup is below:



```
Package Explorer
DrugActivity
    resources
        output
            prediction_waustin_2020-03-08-18-32-57.txt
            prediction_waustin_2020-03-08-19-11-47.txt
            prediction_waustin_2020-03-08-19-13-07.txt
            prediction_waustin_2020-03-08-19-14-27.txt
            prediction_waustin_2020-03-08-20-12-24.txt
            prediction_waustin_2020-03-14-14-05-03.txt
            prediction_waustin_2020-03-14-14-09-55.txt
            prediction_waustin_2020-03-14-14-19-08.txt
            prediction_waustin_2020-03-15-16-42-25.txt
        1582371142_761199_train_drugs.dat
        1582371142_7687795_test.dat
        1582371142_77237_example_entries.txt
    src
        cs584
            __pycache__
            project2
                __pycache__
                __init__.py
                common.py
                constants.py
                data_balancing.py
                drug_record.py
                feature_reduction.py
                information_gain_reducer.py
                legacy_naive_bayes_model.py
                naive_bayes_classifier.py
                run_decision_tree.py
                run_naive_bayes.py
                run_neural_network.py
                tune_decision_tree.py
                tune_naive_bayes.py
                tune_neural_network.py
                utilities.py
            __init__.py
```

The project structure is typical for Python applications, with the **`src/`** directory being the main source directory. All my code is in the **`cs584.project2`** folder.

All input files for the project (training, test, and format) as well as my generated output files are in the **`resources/`** directory. In my submission, I am including the contents of this directory, but excluding the provided input files, as these are somewhat large.

For this assignment, we were required to implement Naïve Bayes algorithm for classifying records representing drug molecules, and in addition, we were also required to try using decision trees and neural networks for the same task. Therefore, I have 2 source files for each of these techniques: one for trying new things out and parameter tweaking (named `tune_<method>.py`), and another for creating a submission file named `run_<method>.py`.

To sumarize, the files below are my main "playground" files, typically with a function for each different thing that I tried out over the course of the project.

- `tune_naive_bayes.py`
- `tune_decision_tree.py`
- `tune_neural_network.py`

And these files are for creating submission files. In general, each function corresponds to different times that I wanted to create a submission file for Miner, after performing cross-validation in the "tuning" file.

- `run_naive_bayes.py`
- `run_decision_tree.py`
- `run_neural_network.py`

Note that my code for this project is also available on GitHub. The URL is:

https://github.com/william-r-austin/DrugActivityPrediction

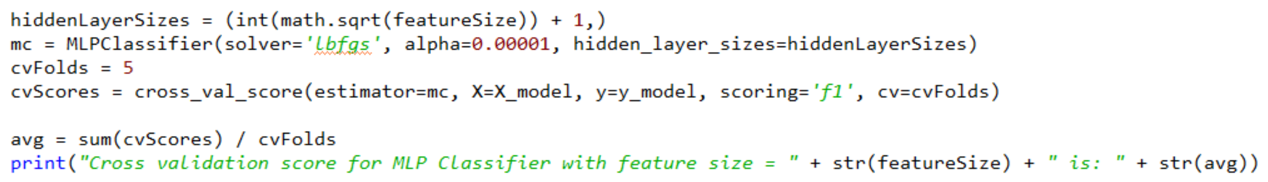
4. Approach and Parameter Selection

I spent significant amounts of time on this project tuning parameters and experimenting with different methods that may yield improved accuracy. The vast majority of these efforts were unsuccessful. I'll highlight the main techniques that I experimented with, mentioning what did and did not work, and if applicable, where to find the implementation in my code.

1. **Data Representation**
   The first task that I completed for this project was to parse the input file and store it in a format that could be easily processed. Because I chose Python, a numpy array is the obvious choice, but as was the case in the first project the data size is somewhat large. Given that each of the 800 records has 100,000 features, storing the data explicitly isn't quite feasible. Fortunately, the input data is very sparse, so I chose to use the `scipy.sparse` package and the `lil_matrix` class to store the data. This class stores a list of non-zero cells, sorted by row and then column. In addition, the vast majority of numpy array operations are supported. Given that the data sets are extremely sparse, this was a sensible choice.

2. **Support for K-Fold Cross Validation**
   In the first project, I used Java and had to write my cross-validation code from scratch. One of the extremely convenient things about using Python and sklearn for this project is that there are libraries and helper functions for executing K-fold cross validation on a trained model. Indeed, I did end up using the `sklearn.model_selection.cross_val_score` function in

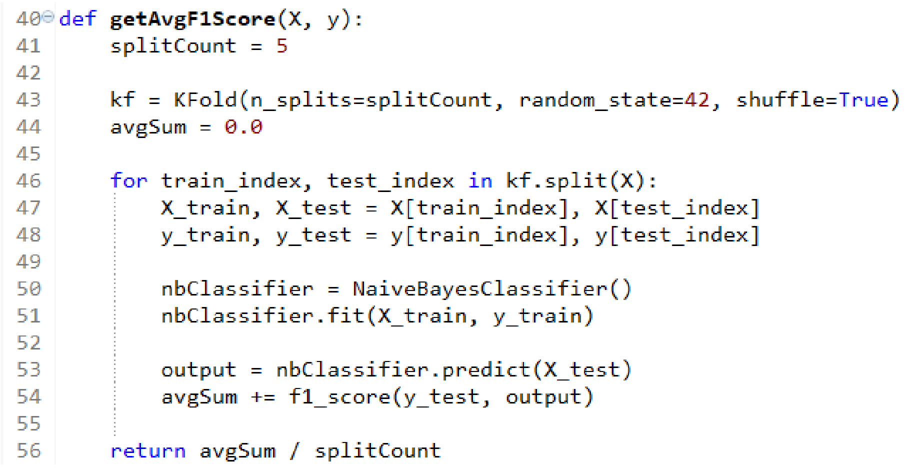
many cases to determine the cross validation score for a classifier. For example, the code below shows my code for training and evaluting a neural network model.



```
hiddenLayerSizes = (int(math.sqrt(featureSize)) + 1,)
mc = MLPClassifier(solver='lbfgs', alpha=0.00001, hidden_layer_sizes=hiddenLayerSizes)
cvFolds = 5
cvScores = cross_val_score(estimator=mc, X=X_model, y=y_model, scoring='f1', cv=cvFolds)

avg = sum(cvScores) / cvFolds
print("Cross validation score for MLP Classifier with feature size = " + str(featureSize) + " is: " + str(avg))
```

In addition, I also used the **`sklearn.model_selection.KFold`** class in many cases as well, where I needed more control. For example, the code below shows my F1 score calculation for my **`NaiveBayesClassifier`** class I created because I did not fully implement both **`BaseEstimator`** and **`ClassiferMixin`**.



```
def getAvgF1Score(X, y):
    splitCount = 5

    kf = KFold(n_splits=splitCount, random_state=42, shuffle=True)
    avgSum = 0.0

    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        nbClassifier = NaiveBayesClassifier()
        nbClassifier.fit(X_train, y_train)

        output = nbClassifier.predict(X_test)
        avgSum += f1_score(y_test, output)

    return avgSum / splitCount
```

3. Implement a custom Naïve Bayes Classification model

Initially, to test the core functionality of reading the input files and processing an output file, I bootstrapped my classification code using the **`sklearn.naive_bayes.BernoulliNB`** class. However, implementing our own classifier for the Naïve Bayes algorithm was focus of this project, so it was the next task that I tackled.

The implementation I created is in the **`naive_bayes_classifier.py`** file, in the **`NaiveBayesClassifier`** class. As is typical, there are 2 main operations in this class: **`fit()`**, and **`predict()`**. The lists below show a high-level view of what happens in each of these functions.

Overview of **`NaiveBayesClassifier.fit()`**

- Break the training set by class label. This gives us 2 data sets: one for the records representing active molecules and another for the inactive molecules.
- For each of these 2 data sets, reduce them to a single row by summing each column. This structure represents the number of records in the data set with a positive indicator for each feature.
- For each feature, compute a smoothed log-likelihood for the conditional probability that feature is 0 or 1, given the class. This represents $P(X_k = 0 \mid c)$ and $P(X_k \mid c)$,

and we compute values for every feature in our training set. Later, we will show why using log-likelihood gives better results than storing raw probabilities.

- Store the computed array of log-likelihood values as an instance variable so it can be reused later.

Overview of `NaiveBayesClassifier.predict()`

- When we need to make a prediction for a new record, we will use the precomputed array from the fit() function to determine which class is more likely.
- We use the fact that all feature values are 0 or 1. Therefore for each class we can multiply the input record by the $P(X_k = 1 \mid C)$ array and it will give us the log-likelihood for $P(X_k = 1 \mid C)$, exactly where the features in the record have a value of one. Likewise, we can perform element-wise multiplication of the array $1 - X$ and $P(X_k = 0 \mid C)$. This will give us an array with class likelihood values for just the features where $X_k = 0$.
- By adding these two arrays together, we get a third array (for each class) representing:

$$\sum_{k=1}^{N} log\, P(X_k \mid C\,)$$

- Lastly, we compare these sums between the two classes to see which is larger to make our prediction. However, in our implementation, note that the **`predict()`** operates on a full test set using numpy arrays and the performance is quite good.

One of the key decisions that I made was to use log-likelihoods for my implementation. Note that the basic Naïve Bayes formula is:

$$P(C \mid X) \sim P(C) \prod_{k=1}^{N} P(X_k \mid C)$$

However, this is a problem for datasets that contain many features because probabilities are typically less than 1, and the product will tend to 0 very quickly. Computers have limited numerical precision, so this can lead to many issues.

Fortunately, we can use the rule that $\log(a * b) = \log(a) + \log(b)$, so we can take the log of both sides and compute a sum (shown above) instead of a multiplication. This is much more numerically stable and allows us to handle large numbers of features without any problems.

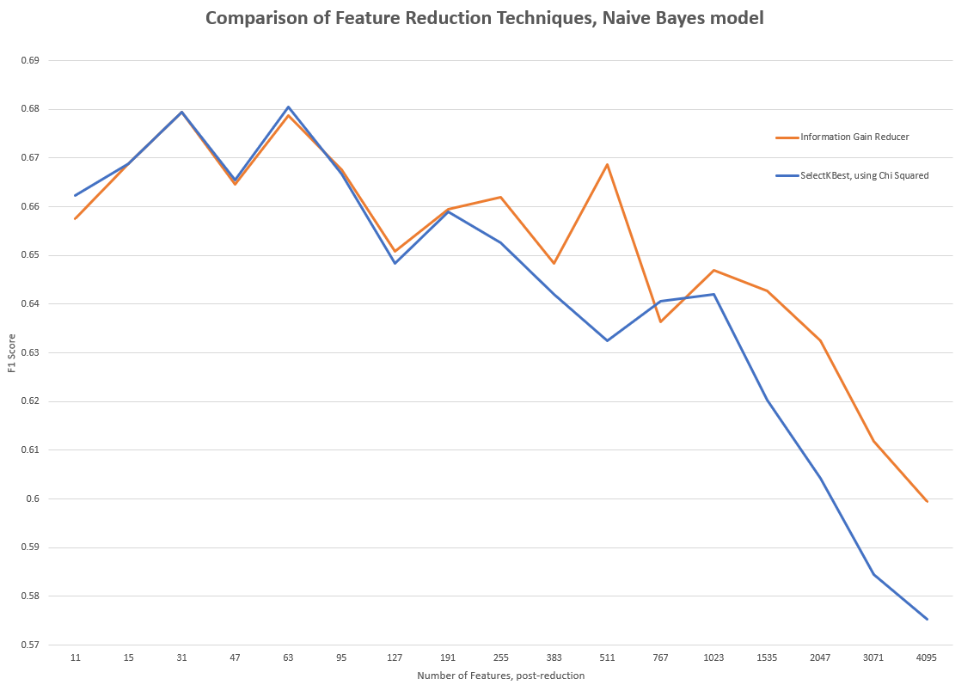
Another design choice I made was to drop the prior probabilities, $P(C)$ for each class, because we don't know the distribution of the target dataset, and depending on resampling, they may be wrong anyway. In addition, assuming we have many features in each record, combining class likelihoods will overwhelm the prior, which is a single term. Dropping the prior in this way is common, and there is an option for it in the **`sklearn.naive_bayes`** models.

4. **Choose Dimensionality Reduction Techniques**

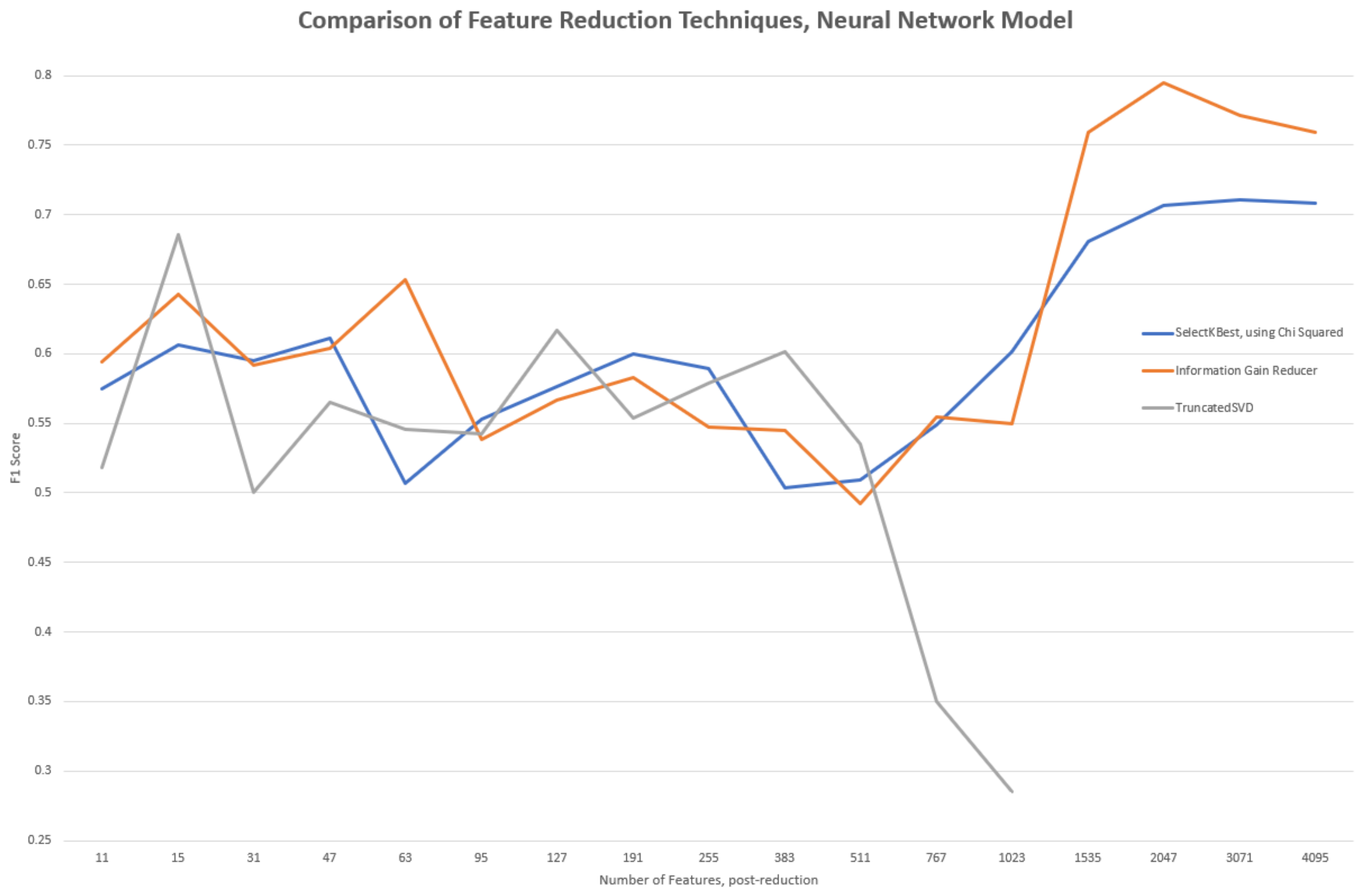
Unfortunately, due to the way I implented Naïve Bayes, my custom classifier does not operate on sparse arrays. Therefore, it is essential to cut down the number of features in the data set before trying to use it with a classifier. There are 3 main techniques that I used for this purpose:

1. The **`SelectKBest`** class, in the **`sklearn.feature_selection package`**, with **`score_func='chi2'`**. This method works by using the chi-squared statistical test to compute which features are most relevant for class prediction. It runs quickly and gave me the best results.
2. My custom I**`nformationGainReducer`** class, implemented in the **`information_gain_reducer.py`** file. This class uses the concept of information gain (similarity to entropy in decision trees) to rank features by importance. It was motivated because I found the **`score_func='mutual_info_classif'`** option for **`SelectKBest`** to be too slow. It has the addional benefit that a global ordering is saved, so the data only needs to be fit once, and any subsequent transformation for any k value (assuming a top-k feature count for reduction) are very fast because we can call the **`resize()`** function.
3. Lastly, I investigated the **`TruncatedSVD`** class from the **`sklearn.decomposition`** package. Note that this method is somewhat different than the first 2 because it operates on the training set but is not dependent on the training labels. Rather, it simply tries to compress the data into a smaller number of features. In addition, the resulting array is real-valued, which is no longer suitable for our Naïve Bayes implementation. This algorithm also runs relatively quickly for values of k that were less than 1000, but somewhat slowly for anything larger. Overall, the performance of this method was somewhat less suitable than the first 2 methods.

The graph below shows the relative performance of the **`SelectKBest`** class and my custom **`InformationGainReducer`**.



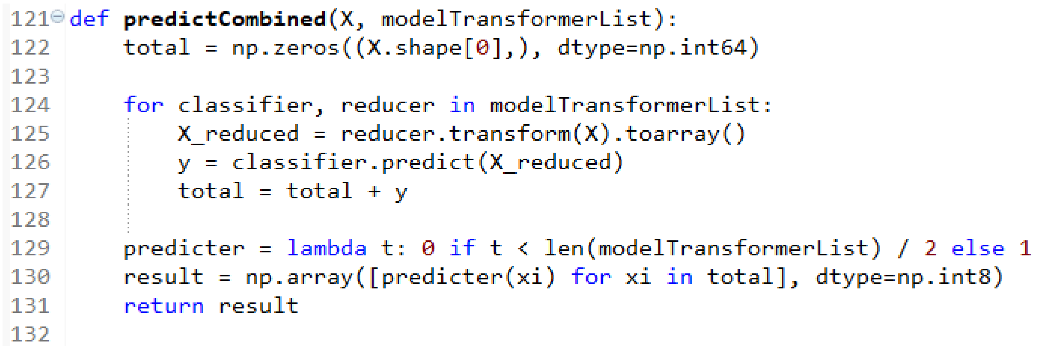
The next graph shows the result for all three techniques when training my neural network model.



5. Dealing with an Imbalanced Data Set

I would say that the biggest challenge with completing this project was determining how to deal with the imbalanced data set. Over the course of working on the assignment, I tried several different techniques:

- The first approach I tried was using **`RandomOverSampler`** class from the **`imblearn.over_sampling`** package, which randomly resamples from the minority class until we have a balanced data set. However, I quickly realized that this leads to problems with cross-validation because there are many copies of the minority class in both the training and test sets, so the model is very overconfident. The low Miner score of my first submission (0.24) was due to this mistake.
- Next, I tried undesampling using the **`RandomUnderSampler`** class, also from the imbalanced-learn library. This improved my results (and reduced the train/test discrepancy), but the fact that we are throwing away 90% of the inactive samples in the training set means that we could be creating better models by using more data.
- Due to the limitations of just using the RandomUnderSampler for training, I decided to expand it by training multiple models, where each would have an undersampled, but balanced data set. Then the final, aggregated predictions would use a majority voting strategy to set the final value from the sub-models. Sample code from the **`common.py`** file to accomplish this is shown below:



```python
def predictCombined(X, modelTransformerList):
    total = np.zeros((X.shape[0],), dtype=np.int64)

    for classifier, reducer in modelTransformerList:
        X_reduced = reducer.transform(X).toarray()
        y = classifier.predict(X_reduced)
        total = total + y

    predicter = lambda t: 0 if t < len(modelTransformerList) / 2 else 1
    result = np.array([predicter(xi) for xi in total], dtype=np.int8)
    return result

```

In my code, I refer to this as a multi-model. This strategy gave me the best overall result, and my last Miner submission is based on a multi-model.

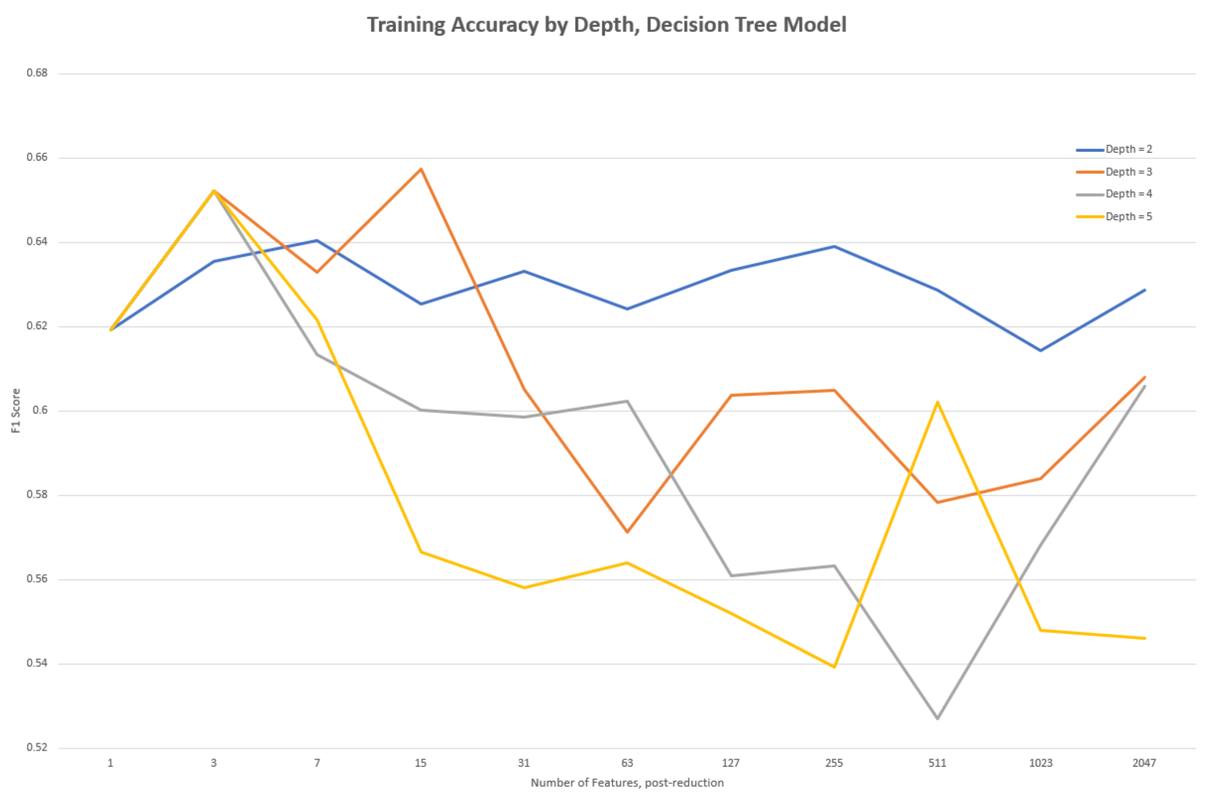
- To fix the imbalanced data problem without having to deal with the complexity of dealing with sub-models, I wrote my own custom over-sampling class to generate new samples from the minority class where each feature is constructed independently. Therefore, if we assume feature independence, this method generates samples that are representative of the minority class. However, I found that this method did not give good results and still produced overconfident. The code is in the **`FeatureIndependentOversampler`** class in the **`data_balancing.py`** file.
- I also briefly experimented with **`imblearn.over_sampling.SMOTE`** class. I did not have time to re-run many of my experiments, but I found it performed comparably to my multi-models, and certainly better than my custom over sampler.

6. Implementing Classifiers for Decision Tree and Neural Network Models

The next task for this project was to implement a classifier using the decision tree and neural network models. Although these were not my best results, a brief discussion is below:

<u>Decision Trees</u>

Naturally, I used the **`sklearn.tree.DecisionTreeClassifier`** class. However, this class has quite a bit of complexity built into it, and I found the task of finding suitable parameters to be quite difficult. The **`max_depth`** parameter is what I adjusted and tried to experiment with the most, as the chart below shows.



There are many other parameters, such as `min_samples_split`, `min_samples_leaf`, `max_leaf_nodes`, `min_weight_fraction_leaf`, and `min_impurity_decrease` that I did not have time to fully explore and tune.

<u>Neural Networks</u>

For this task, I used the `sklearn.neural_network.MLPClassifier` class. The main architecture that I experimented with was to have a single hidden layer with a size of the square root of the input size after feature reduction. Unfortunately, I did not get good results with this method. I felt that the data set would not be large enough to support the training for another layer with more nodes. I did run a few tests to tune alpha after optimizing the feature reduction size and found that modifying alpha did not substantially change the F1 score. Therefore, the best performance I got on a Miner submission generated by this method was 0.59. In addition, it was also the slowest method for training.

7. Lessons Learned and Other Techniques

Briefly, I'll outline a few of the takeaways from this assignment:

- Dealing with an imbalanced data set is difficult. Continuing to experiment with the SMOTE and ADASYN methods would probably be beneficial.
- I briefly tried several other methods using pre-packaged classifiers in the sci-kit learn library. This includes:
    - SVM Classifier: `sklearn.svm.SVC`
    - Random Forest Classifier: `sklearn.ensemble.RandomForestClassifier`
    - K-Nearest Neighbors: `sklearn.neighbors.KNeighborsClassifier`

Unfortunately, none of these methods give better results than I had already achieved with Naïve Bayes. Perhaps with further tuning, they could be competitive.

- When trying to implement new ideas, it makes sense to try to conform to the established standards by the sci-kit learn library. This allows your code to be plugged in and used by existing functions and classes in the library, making life simpler. However, I found this somewhat difficult because I am not as familiar with Python as I am with Java.