Kyle Jackson and Robert Truong

CS 584

HW2

**HW2: Drug Activity Prediction**

Team name: econguy19

Rank & Public Score (as of 10/3/2016): 12, 0.73

**Approach and Methodology**

There were two notable traits about the data given in this project. First, there were many features, totaling 100,000, but each record had relatively few of them. Only present features were stored, in a space-delineated list after the label. Second, the distribution of labels was heavily skewed. Around 90% of the records were negative, and 10% positive. So our approach had to take these two things into account.

The overall format of our approach could be broken down into four parts. First, we read in data, and processed it into a scipy sparse matrix. Next, we used feature selection to boil down the feature set into its most useful components. Third, we used undersampling to balance the data reduce the number of features. After that, we ran it through a classifier.

The train and test files were first read into two separate pandas DataFrame objects. The format of these are an *ns* *x* *mf* matrix, where *n* represents the sample, *s* represents the sample index, *m* represents the features, and *f* represents the specific feature. At this stage the training data has a dimension of *800 x 2*, where the first column is the label and the second column contains the list of features as a string, for that specific sample. Similarly, the test data is a DataFrame of dimension *350 x 1*, where the one column contains the features all in one line.

Since the characteristics of the compounds are integers that are all in one line, analyzing these data is quite difficult, particularly for the classification task at hand. The integers represent features that are present in each sample, so we binarize the features into a sparse matrix by looping over each list of features, and putting in 1 at the coordinate (record number, feature number - 1). The -1 is just because the features are 1-based while the matrix is 0-based. The new structure preserves the same number of samples, but each feature is in its own column.

We tried several methods for feature selection, and made particular use of scikit-learn’s feature\_selection module. One that we tried was PCA, which is actually a dimensionality reduction algorithm that looks for, basically, the most important parts of a matrix. What PCA does is it turns a matrix into a product of three matrices, and the middle matrix contains a value corresponding to each “component” of the matrix, in order of their importance. One can then remove all but the top few components, and this gives a smaller-dimension matrix that should estimate the full matrix well. But this gave poor results, so it probably didn’t fit the problem.

We also made use of SelectKBest, which selects the *k* best features based on a designated scoring function. In our implementation, we selected the features with the highest chi-square statistic. The chi-square test measures dependence between stochastic variables, so using this function “weeds out” the features that are the most likely to be independent of class, and therefore irrelevant for classification. In order to determine the number of features to keep, we used 10-fold cross-validation and found that 225 features was optimal.

We finally tried to use a function from scikit-learn called RFE, also known as recursive feature extraction. It uses a different supervised learning algorithm (in our case, we used linear Support Vector Classification) that has a matrix variable called “\_coef”. It runs the data, looks at the matrix, and takes out the features that have the smallest coefficients (in absolute value). By default, it removes one feature each time, but we set it to remove 1000 each since doing 100000 iterations wasn’t feasible timewise. It then runs the algorithm again, removes more features, and so on until only the *n* features you requested remain.

When we ran cross-validation with it, it gave a high mean validation score compared to SelectKBest/chi2 (~0.8266 vs. ~), but it didn’t match the test data when we uploaded it to the server, so this algorithm was likely overfitting the data, so we elected to just stick with chi-squared.

For the undersampling, we used a library called imblearn (imbalanced-learn), which has several methods to do so. We did not really know which one of these was best, so I made a graph of all of them on the same set of data on one axis, and the other axis had the number of features to keep from the chi-squared feature selection (see Sheet 1 in the enclosed Excel sheet). We determined that the edited nearest-neighbor was the best performing undersampling algorithm. This algorithm essentially removes samples from the negative (0) class if there is a neighbor from the positive (1) class. This leaves a cluster that is purely positive and reduces false negative classifications.

In general, each undersampling algorithm reached a maximum somewhere between 175 and 250. The number of features chosen skew near 200, because when we did preliminary calculations with cross-validation, that seemed to be where the numbers roughly maxed out. The largest CV value was obtained with (EditedNearestNeighbours, k = 225), but none of them were significantly better than the others (including random undersampling), so the choice of undersampling algorithm must not be terribly significant.

Finally, for our choice in classifier, we chose Naïve Bayes. We used another library called TPOT, which uses genetic programming to solve for the best pipeline. Basically it does a bunch of runs for a bunch of feature reduction and classifier choices, “kills off” the weaker entries, then in the next generation it will add random “mutations” and then the thing repeats for more generation, roughly speaking. In general, our TPOT results tended to be either Bernoulli Naïve Bayes (which is just Naïve Bayes for binary data, which is what we have here) or varying Trees, which had high cross-validation scores on the training set, but poor scores on the leaderboards, which probably suggests overfitting again. We also tried using the Multinomial Naïve Bayes classifier, but the results were much poorer when compared to Bernoulli Naïve Bayes. In addition, we performed 10-fold cross-validation several times to determine the best alpha parameter, which is an additive smoothing parameter. We chose .75, which had a reasonably high mean (though not the highest) and smaller standard deviation (see Sheet 2 in the excel sheet for the data).

Naïve Bayes basically works as follows. You want to choose the best class *C* for a given set of features *Ai*. You can use Bayes’ Theorem to rewrite this as:

If each *A1, A2*… *An* is independent (which is a big assumption, but that is why it’s naïve), then this can be rewritten as (product over all *i*):

The numerator can be calculated, and the denominator does not change if the class does, so choosing the class that maximizes (product over all *i*) P(*Ai*|*C*) \* P(*C*) will give the best label. Where the Bernoulli name comes from is that each P(*Ai*|*C*) can be written as a Bernoulli trial p(1-p), where p is the probability that *Ai* manifests in members of class *C*.