SYSC/BIOM 5405

Pattern Classification and Experiment Design

Project Report

Submitted by Group 2

Arshad Mehtiyev

Dhiraj Bhattacharjee

Joshua Wilson

Juhi Khalid

Instructor: Prof. James Green

December 2021

Contents

List of Figures

List of Tables

1 Introduction

A CSV data file with 9479 rows of drug-target pairs was provided. It has 24*14= 336 columns of features using reciprocal feature extraction. The "KIBA" column contains continuous Kiba score and "Labelâ column contains binary TRUE/FALSE indicating whether drug-target pair is viable where Kiba score 12.1 indicates TRUE.

The problem is to develop a classifier to predict whether the label is TRUE or FALSE.

2 Experiment Design

For the experiment design, the whole dataset is first randomly shuffled and 10% is held out for testing and remaining 90% is used for training the neural network. The training set is cleaned and preprocessed, after which feature extraction is carried out. Neural network requires the data set to be balanced and SMOTE is used to achieve the same. The balanced training dataset is then fed to three component classifiers (three neural network) with different hyperparameters. Their predictions are combined by voting as the part of ensemble learning to get the final prediction. Finally, the test set is fed to the final model to investigate the performance of the model with future data.

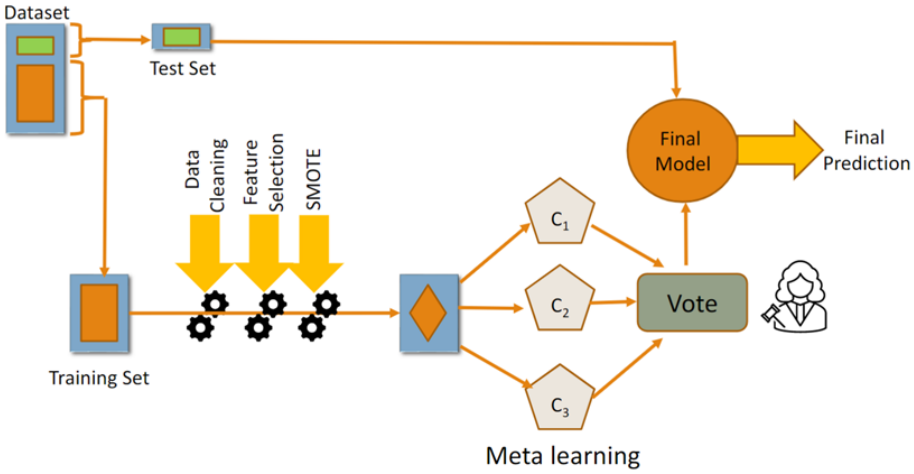


Figure 1: Experiment Design Flow

3 Preprocessing

Inspecting the data, it was seen that there are no missing values. As shown in Figure 2, there are 23155 TRUE and 86324 FALSE instances. Data preprocessing is an important

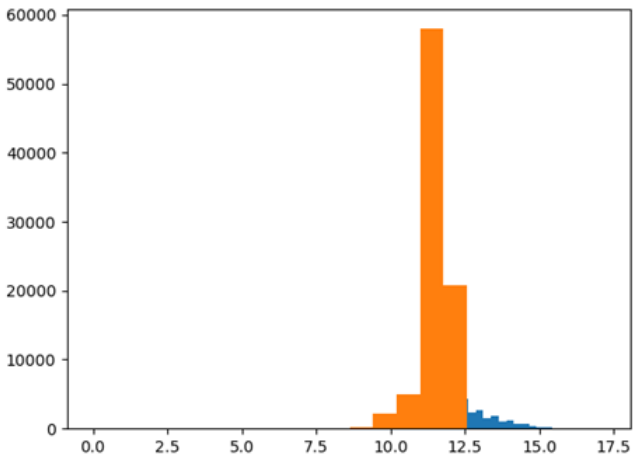


Figure 2: Data classes with respect to Kiba score

part of machine learning. It ensures relevant data samples are used and redundant features are removed. The following subsections give the steps in preprocessing.

3.1 Data splitting

The data was split into training set(90%) and testing set(10%) to allow for hold out test to assess the efficiency of the final classifier. Stratified splitting was carried out to maintain the class imbalance in test data.

3.2 Class imbalance

The provided data has a high class imbalance. It is imperative for the training set to be balanced, otherwise the dominant class will be predicted more and rare class will be ignored. In addition, neural networks require equal samples from all classes otherwise, it will fail to learn the patterns in the rare class. To take care of this issue, SMOTE(Synthetic Minority

Oversampling Technique) was applied to the smaller class. It generates synthetic data at a randomly selected point between 2 of the k nearest neighbours in the minority class. Applying SMOTE, the number of positive and negative instances in the training data were made equal.

3.3 Removing Outliers

Outliers in the training set lead to overfitting and we might fail to generalize blind test data. So, outlier data samples were determined based on z-score. Samples with z-score greater than certain threshold (considered as 500 in this experiment design) were removed.

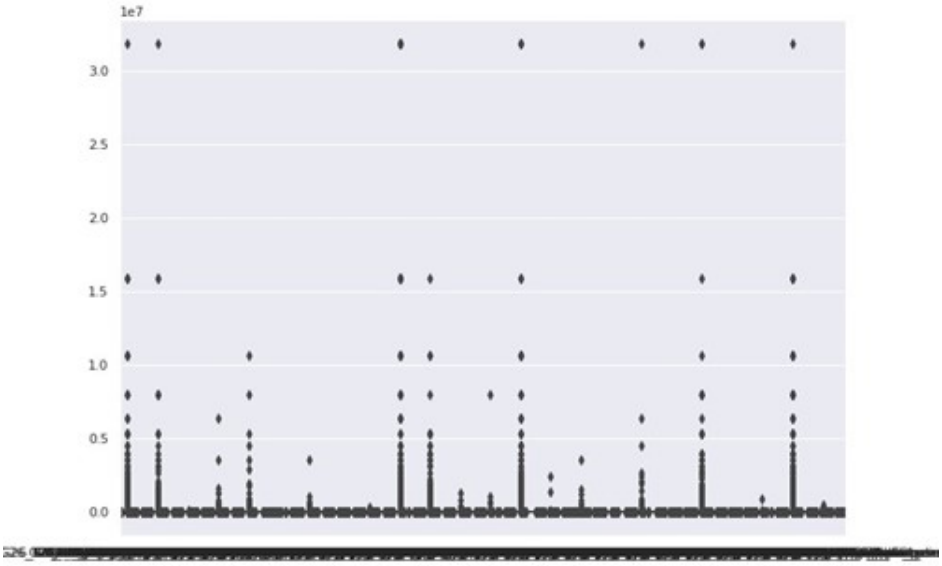
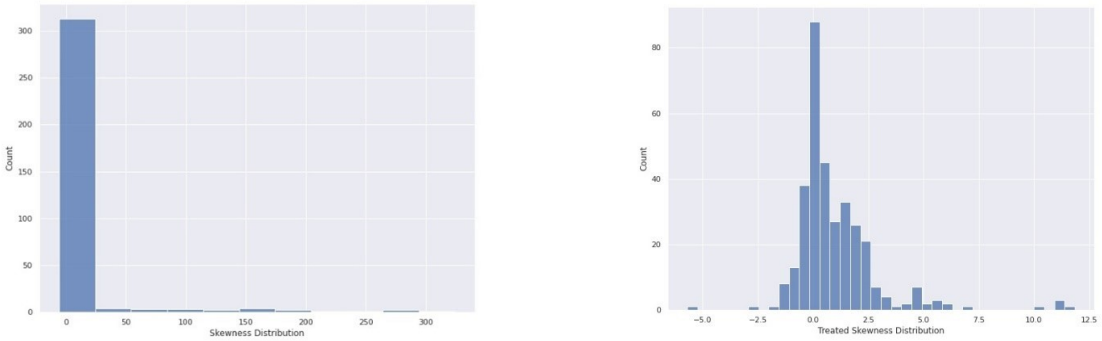


Figure 3: Boxplot of all 336 features for visualization and outlier detection

3.4 Skew Adjust

High skewness can affect the statistical performance, as it might act as an outlier. In addition, having Gaussian distribution makes dataset easy for several methods such as t-test, F-test, ANOVA etc. Hence the features were made closer to Gaussian distribution, instead of



(a) Before (b) After

Figure 4: Distributions before and after skewness treatment

removing them. After analysing skewed features, we got 23 of them and all were Gxx_ARRO features. To treat them, log() function was applied. Figure 4 shows the difference in skewness before and after.

4 Feature Selection

Several feature selection methods were considered. They are mentioned below.

4.1 Correlated feature removal

For Neural networks, it is assumed that all the features are independent. To satisfy this, the correlation between each pair of features in the data are computed. If a pair of features have correlation higher than 0.9, only one of the pair is kept and the other feature is dropped.

4.2 Principal Component Analysis(PCA)

PCA was applied to the data and only components which most explains the variance in the data is kept. It did not much aid the separability of classes. This is due to the fact that PCA is an unsupervised method which uses eigenvectors of the data to find the new principal axes to reduce dimensionality.

4.3 Selecting Features based on Target

Along with the need to remove features which are highly correlated to each other, it is also important to keep those features which are highly correlated to target variable (i.e. "Label"). For this , SelectKBest was tried along with chi2 and f_classif(Anova).

After using different combinations of the listed methods on a trial and error basis, finally 248 features after correlation removal were selected.

5 Classifier design

After feature selection, there were 248 features to feed into 248 inputs of to each of the 3 neural network classifiers. Only 1 hidden layer was used from all 3 models. The number of hidden nodes used by each of the classifiers are 60, 40 and 20.

5.1 Choice of Method

Our suggested methods for the project were: SVM, KNN and FFNN. And we were confirmed for FFNN.

Feed forward neural network(FFNN) is one of the popular ML models. It is well suited for problems with linearly inseparable data. They can create arbitrary decision regions with even a small number of training data. Also, FFNN comes with several optimization parameters which helps fit any type of boundary as required for the problem.

After data exploration, it was visible that positive and negative classes are not linearly separable and there is high class imbalance. Chosen method seemed to fit the problem.

5.2 Implementation

PyTorch was used alongside several python implementations of various statistical algorithms. Pytorch was selected as it presented the lowest barrier to an initial implementation and pro-

vided more concise way of defining the model and data pipeline. Alternative implementations we could have made use of were TensorFlow and Keras.

The 2 main components of our model's implementation are the DTIDataset class and the NeuralNet model itself. The DTIDataset is a custom implementation of PyTorch Dataset. This class provides tooling for splitting data, permitting us to perform holdout or cross-validation tests. We made use of this class to perform all pre-processing as well. This permitted us to develop the model agnostic to how data is processed before being provided with the dataset.

Since the purpose of this implementation is experimental it is useful to be able to rapidly adjust the parameters of the experiment. As such, the constructor for the DTIDataset takes several parameters which determine which steps should be taken in pre-processing.

```
def __init__(self,
             csv_path,
             validation_path,
             useKiba=False,
             remove_outlier=False,
             outlier_threshold = 500,
             feature_select=False,
             princ_comp=False,
             num_features=168,
             feature_selection_metric=chi2,
             holdout_size=0.1,
             skew_adjust=False,
             smote=True,
             corr_remove=True):
```

Figure 5: Code section: preprocessing options

In effect, the final model would use only a few of the pre-processing techniques found to be most effective.

The second portion of our implementation is the model itself. Our model is defined as an implementation of nn.Module, from the PyTorch library. We define a NeuralNet class which takes an input size, hidden size, and output size as parameters because it is an implementation of a 3-layer neural network. To use this model, we must define the nodes in the network as well as define the forward method. This method describes the behaviour of the network, i.e. how data passes from the input to the output in a forward pass.

The 2 design decisions made here are: how many layers and connections should exist in the network as well as which non-linearity should be used. An example of a possible forward method might look as follows:

```
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.input_size = input_size
        self.l1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.LeakyReLU()
        self.l2 = nn.Linear(hidden_size, num_classes)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        out = self.sigmoid(out)
        return out
```

Figure 6: Code section:ffnn parameters

Finally, to train the network we decide on which criterion is suitable to evaluate the model, we decided on an optimizer, and we made use of other training techniques such as using a varying learning rate and momentum. We decided on a number of epochs over which to train the model we then made use of the dataset class to sample example data top show to the network. Then, given the criterion and the optimizer, we can update the weights in the network.

We implemented all this in a Jupyter notebook allowing for easy prototyping and reporting. From our trained model we can tried to make predictions and evaluate the performance of our implementation. More evaluation will be described in later sections.

5.3 Training Techniques/ Hyperparameters Used

While training the model we faced a couple challenges, the first being that the model can be computationally expensive. This meant that choice of hyperparameters was limited by the time it would take to train the model. This, in combination with the size of the dataset meant that we could not afford to try every possible combination of hyperparameters. Given

this, an attempt was made to intuitively reduce the computational cost of training the model while also trying to maintain a performant model. For example, making use of stochastic gradient descent, while using a smaller batch size might provide more stable learning, it would also require taking more steps in a given epoch, so any gain in performance had to be weighed against training time. The same can be said about many of the parameters, where it would be possible to perceive an improvement in performance for the case where we simply train more.

Training was done over several epochs, using batch stochastic gradient descent. Making use of the PyTorch implementation of SGD. The Criterion we chose to define our loss function was binary cross entropy, a suitable measure of error in a binary classification problem such as this. PyTorch also provides an implementation for this. Example parameters for training might look as follows:

num_epochs = 10

batch_size = 10

learning_rate = 0.005

This would mean looking at 10 samples from the training set for each learning step and repeating over the entire dataset 10 times. Each step could be made bigger or smaller by adjusting the learning rate.

It was also noted that at some point the loss decrease over training, but the large step size made the result stop converging after a point. As such we added the use of momentum as well as an exponentially decreasing learning rate to allow more fine adjustment when closing in on more optimal weights in the network.

5.4 Meta Learning

Ensemble learning was implemented with 3 neural network classifiers, all trained with the same set of data. The hidden layers used by each of these classifiers are 20,40 and 60. The final prediction is the average of the 3 classifier outputs.

Number of hidden layers	Precision at 50% Recall
60	0.6266 +/- 0.046
40	0.6073 +/- 0.052
20	0.5763 +/- 0.051

Table 1: Performance of individual classifiers

6 Performance while training

After training the data, the held out test set was used to measure the performance of the model.

6.1 ROC curve

Figure 7 shows the ROC curve. The average performance is better than a random classifier with an ROC AUC of 0.64. The class imbalance requires a low false positive rate and so, the leftmost part of the figure is more significant.

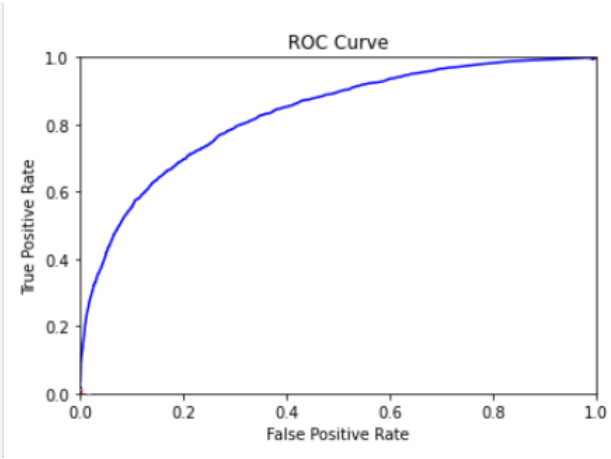


Figure 7: ROC curve

6.2 Precision Recall curve

Figure 8 shows the Precision recall curve. The performance metric required is Precision at 50% Recall, which for the model was measured to be 0.65 at a threshold of 0.69.

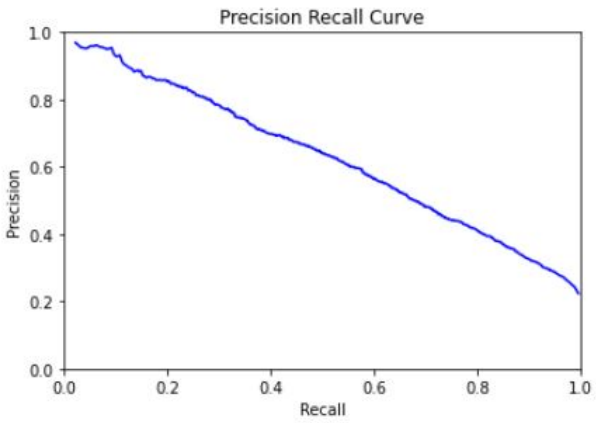


Figure 8: Precision Recall curve

6.3 Expected performance on blind data

In order to predict the future performance of the model, resampling was done 100 times by choosing 1000 samples each time from the test set. A distribution for Precision at 50% Recall was hence plotted as shown in figure 9. Thus, the expected performance for the blind data was expected to be 0.65 +/- 0.05.

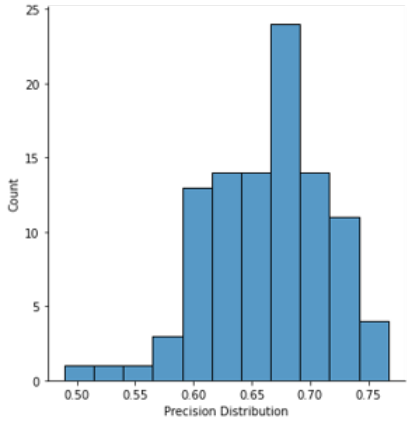
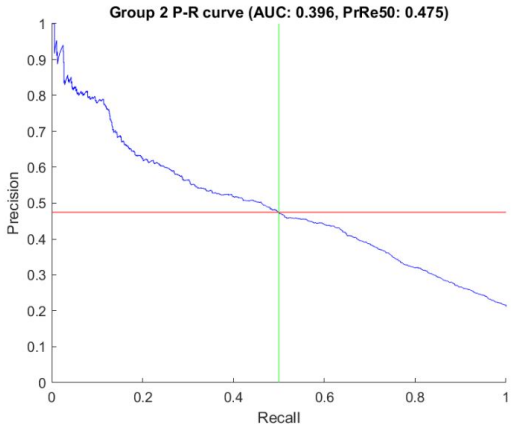


Figure 9: Distribution for Precision at 50% Recall

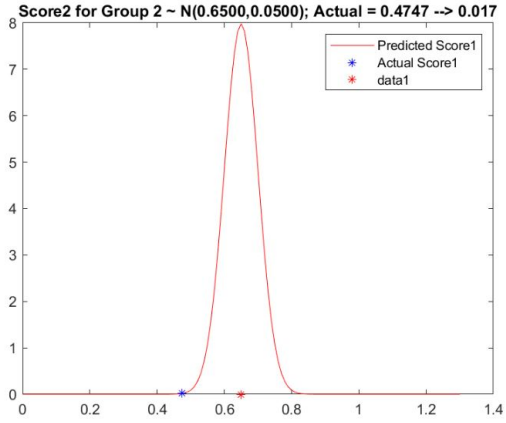
7 Performance on Blind test data

The precision at 50% Recall for the blind data was 0.475. The AUC of the cure was 0.396.

The performance metric lay in the range as predicted in the previous section with a value of 0.017 for the score 2.



(a) Score 1



(b) Score 2

Figure 10: Performance on blind data

8 Critique and Future work

The team enjoyed working collaboratively, and met all the required deadlines. One of the main limitations with the project was the time constraint. The ways in which we would have improved the performance of the model otherwise are:

1. Feature selection: One observation while training was that using more features yielded better results than fewer features, as such the final model made use of 248 of the original 336 features. This translates to at least 248 x hidden_size learnable parameters. As such, it likely played a role in overfitting the model to the training data. For future models fewer features might help the model generalize. Filter methods were used for feature selection. This could have been more refined if wrapper methods incorporating algorithms such as decision trees, KNN were also used.

2. Regression: Only the labels were used for prediction in the model. Neural networks are efficient in implementing regression as well as classification. One way to have exploited the available Kiba scores was to perform linear regression, then classify the results using

threshold 12.1(similar to the available labels)

3.Number of hidden layers: Even though our observations showed a lack of improvement in performance with increase in hidden layers, there is a probability that better results could have been achieved with a tighter set of features combined with more layers as it would have helped capture the non-linearities better.

4.Meta-learning: The final model comprised of 3 neural network based classifiers contributing its decisions. Each network had relatively similar architecture and saw roughly the same information. It might have been advantageous to use many more models and to vary their view of the problem. Several other methods that may have worked better are:

a.Use different sets of training data for different classifiers

b.Use different sets of features for different classifiers

c.Use boosting to improve the performance of the overall classifier

6. Using all training data when training the final model might have been a good option as well to improve performance.

7. Possibility of methodological error: Even though for every run, test split was done by random sampling, we were adjusting parameters based on the test results. Considering the project was done on limited dataset, there is a chance that we intuitively optimized it based on the test set â which could have resulted in overfitting. We could have avoided this by having separate validation and test sets.

9 Conclusion

The team was able to develop a classifier for the presented problem in the limited time. The classifier uses Forward-Neural-Networks to predict the drug-protein interaction. After preprocessing the data, 248 features were selected and fed into 3 neural network classifiers which were combined to give the final prediction. The expected performance was 0.65 +/- 0.05 Precision at 0.5 Recall and achieved a Precision of 0.475 at 0.5 Recall.

References

[1] Lecture notes for SYSC 5405 Pattern Classification and Experiment Design.

[2] Paszke, A. et al., 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems 32. Curran Associates, Inc., pp. 8024â8035. Available at: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[3] Pedregosa, F. et al., 2011. Scikit-learn: Machine learning in Python. Journal of machine learning research, 12(Oct), pp.2825â2830.