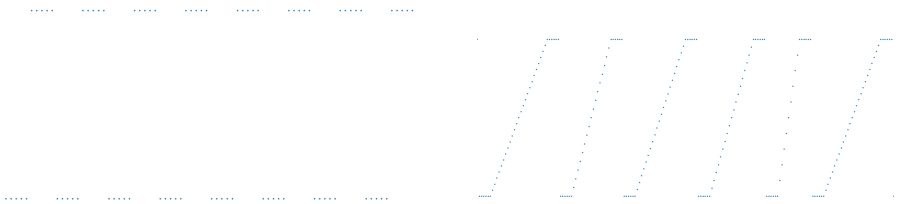
ELEC4830 Final Project

Thomas Narayana Swamy

May 17, 2019

1 Pre-processing

From Labeled Output we can see the a series of zeros is followed by a series of ones. Which shows a close correlation between the labels of Y. So for initial testing, a simple approach was chosen, where all the non-Nan values were extracting which reduces our our number of samples from (16, 13145) to (16, 1500) samples. Figure 1a shows the resulting visualization of the non-Nan values, where we see an alternating series of zeros and ones. This data was then used to train a couple of different classifiers and only produced between 70-76% validation accuracy.



(a) Extracting the Non-NaN values.

(b) Interpolating between consecutive zeros and ones.

Figure 1: Pre-Processing Output

Further pre-processing was conducted as shown in Figure 1b. Data between a series of zeros and ones were linearly interpolated to replace the missing NaN labels making this a regression problem and a threshold is set to classify the final result to either a zero or one. This attempt work very poorly as there does not seem a strong correlation between the zeros and ones and the neural firing between the NaN regions.

The final approach that seemed to have work quite well is to keep a certain amount of historic data when determining the rats state. Figure 2 shows code snippet where the average distance between 2 consecutive non-NaN values. Using this data we can determine how much historic data we can keep without causing overlaps between 2 consecutive non-NaN values . I this case we used kept 25% of the average distance between 2 consecutive non-NaN values which resulted in a significant increase in validation accuracy between 86-96%.

```
Y_raw = mat['trainState']
noNan = numpy.where(numpy.isnan(Y_raw) == False)
Y = Y_raw.take(noNan[1])
Y_raw.shape, Y.shape

counter = numpy.array([97])
for i in range(5, len(noNan[1]), 5):
  counter = numpy.append(counter, noNan[1][i] - noNan[1][i-1])

print("Average Gap:",numpy.mean(counter))
```

```
Average Gap: 39.77333333333333
```

Figure 2: Average Gap Between non-NaN values.

2 Classifiers: Test and Results

4 Classification Methods were used to train on the pre-processed data mentioned above.

2.1 Keras 2 layer Neural Network Classifier

A 2 layer sequential neural network was implemented in this sections. The first layer comprises of 128 neurons and a "relu" activation function. The second output layer is a single output neuron with a "sigmoid" activation function. The system robustness was tested using a 5-fold cross-validation method and the result are shown in Figure 3a. The final accuracy on the validation set is shown in Figure 3b, but the accuracy seems slightly high and over-fitting might be a factor but methods such as the k-fold cross validation technique was used to mitigate over-fitting.

```
from keras.constraints import unit_norm
from sklearn.model_selection import StratifiedKFold
cvscores = []
# define 3-fold cross validation test harness
kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
for train, test in kfold.split(X_train, Y_train):
  # create model
  model = Sequential()
  model.add(Dense(128, input_dim=16, activation='relu'))
#   model.add(Dense(64, activation='relu'))
  model.add(Dense(1, activation='sigmoid'))
  # Compile model
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  model.fit(X_train[train], Y_train[train], epochs=200, batch_size=64, verbose=0)
  # evaluate the model
  scores = model.evaluate(X_train[test], Y_train[test], verbose=0)
  print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
  cvscores.append(scores[1] * 100)
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(cvscores), numpy.std(cvscores)))
```

```
acc: 91.88%
acc: 90.00%
acc: 91.85%
acc: 92.59%
acc: 92.57%
91.78% (+/- 0.94%)
```

(a) K-fold Cross-validation.

```
from sklearn import metrics
print("Accuracy:",metrics.accuracy_score(Y_test, y_pred))
```

```
Accuracy: 0.9166666666666666
```

(b) Accuracy 2 Layer Neural Network.

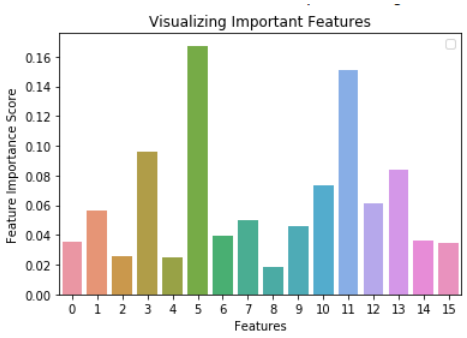
Figure 3

2.2 Random Forest Classifier

A random forest classifier was also implemented. Grid Search technique was used to optimize the hyper-parameters. Then using the classifier was validated and used to extract the key features and rank their importance as shown in Figure 4b. Using this figure we dropped neuron 2,4 and 8 resulting in an increased accuracy from 0.883 to 0.89 shown in Figure 3.

```
param_grid = {
    'bootstrap': [True],
    'max_depth': [80, 90, 100, 110],
    'max_features': [2, 3],
    'min_samples_leaf': [3, 4, 5],
    'min_samples_split': [8, 10, 12],
    'n_estimators': [100, 200, 300, 1000]
}
```

(a) Grid-Search Random Forest Hyper-parameters.



(b) Feature Importance.

Figure 4

```
from sklearn import metrics
print("Accuracy:",metrics.accuracy_score(Y_test, y_pred))
```

```
Accuracy: 0.8833333333333333
```

(a) Before Feature Importance Sort.

```
from sklearn import metrics
print("Accuracy:",metrics.accuracy_score(Y_test, y_pred))
```

```
Accuracy: 0.89
```

(b) After Feature Importance Sort.

Figure 5: Accuracy on Validation Set

2.3 XGboost Classifier

A XGboost classifier was also implemented. Bayesian Search technique was used to optimize the hyper-parameters (Figure 6a), the hyper-parameters found here are optimum as XGboost is GPU optimized. Then using the classifier was validated on the validation set resulting in a 90.6% accuracy which is shown in Figure 6b.

```
xgb_opt = BayesSearchCV(
    XGBClassifier(tree_method='gpu_hist'),
    {
        'max_depth': (3, 15),
        'gamma': (1e-6, 1e+1, 'log-uniform'),
        'learning_rate': (0.01, 0.4, 'log-uniform'),
        'min_child_weight': (1, 10),
        'subsample': (0.5, 1.0, 'log-uniform'),
        'colsample_bytree': (0.5, 1.0, 'log-uniform'),
        'n_estimators': (100, 1000)
    },
    n_iter=32,
    random_state=42,
    cv=3
)
```

(a) Bayesian-Search XGboost Hyper-parameters.

```
y_pred = xgb_opt.predict(X_test)
from sklearn import metrics
print("Accuracy:",metrics.accuracy_score(Y_test, y_pred))
```

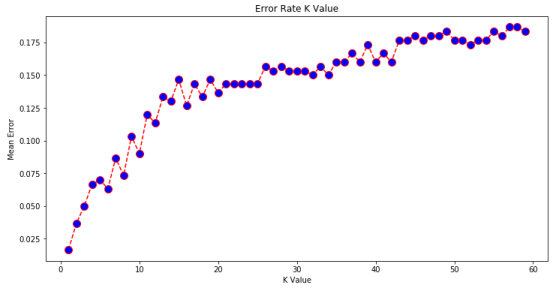
```
Accuracy: 0.9066666666666666
```

(b) Accuracy XGboost.

Figure 6

2.4 K-Nearest Neighbor Classifier

A KNN classifier was also implemented. The classifier's performance was evaluated for different K values as shown in Figure 7a. Using the graph a suitable value of K was selected near the plateaued in order to prevent over-fitting. Then using the classifier (with a K value of 31) was validated on the validation set resulting in a 84.6% accuracy which is shown in Figure 7b.



(a) Mean Error vs K value.

```
y_pred = classifier.predict(X_test)
from sklearn import metrics
print("Accuracy:",metrics.accuracy_score(Y_test, y_pred))
```

```
Accuracy: 0.8466666666666667
```

(b) Accuracy KNN.

Figure 7

2.5 Conclusion

Below we can see the confusion matrices for each classifier mentioned above. The performance of all the classifiers are as expected and the distribution of the matrices are fairly similar. Further tests need to be conducted to evaluate which classifier performs the best.

```
[[139  21]
 [  4 136]]
```

(a) Neural Net.

```
[[128  32]
 [  1 139]]
```

(b) Random Forest.

```
[[136  24]
 [  4 136]]
```

(c) XGboost.

```
[[120  40]
 [  6 134]]
```

(d) KNN.

Figure 8