CS 334: Homework #5 Solutions

1. **(5+10+10=25 pts) PCA**: The full code can be viewed in `q1.py`

   (a) We use StandardScaler and combine that with the unregularized logistic regression.

```
def normalize_feat(xTrain, xTest):
    """
    Normalize xTrain and xTest
    """
    stdScaler = skp.StandardScaler()
    stdScaler.fit(xTrain)
    return stdScaler.transform(xTrain), stdScaler.transform(xTest)


def unreg_log(xTrain, yTrain, xTest, yTest):
    logR = sklm.LogisticRegression(penalty='none',
                                   solver='saga',
                                   max_iter=10000)
    logR.fit(xTrain, yTrain)
    # predict the probabilities
    yHat = logR.predict_proba(xTest)
    # calculate the ROC
    fpr, tpr, _ = skm.roc_curve(yTest, yHat[:, 1])
    return fpr, tpr
```

   (b) For PCA, we first use the training data to determine the number of components needed to fit 95% of the variance, then we can transform both train and test according to the number of components. This turns out to be 9 components.

```
def run_pca(xTrain, xTest):
    # set the shape to be the max
    pcaModel = skd.PCA(n_components=xTrain.shape[1])
    pcaModel.fit(xTrain)
    # calculate number of components to get to 95%
    expVar = pcaModel.explained_variance_ratio_
    totVar = np.cumsum(expVar)
    k = np.argmax(totVar > 0.95) + 1
    # refit it to this value
    pcaModel = skd.PCA(n_components=k)
    pcaModel.fit(xTrain)
    return pcaModel.transform(xTrain), pcaModel.transform(xTest), pcaModel.components_
```

   (c) For the ROC curves, you need to predict the probabilities and then calculate the false positive rate and true positive rate and plot both.

```
    # normalize the features
    nxTrain, nxTest = normalize_feat(xTrain, xTest)
    fpr[0], tpr[0] = unreg_log(nxTrain, yTrain, nxTest, yTest)
    # run pca and train another logistic regression
    pcaTrain, pcaTest, pcaComp = run_pca(nxTrain, nxTest)
    pcaDF = pd.DataFrame(pcaComp, columns=feats)
    pcaDF.to_csv("pcaComps.csv")
    fpr[1], tpr[1] = unreg_log(pcaTrain, yTrain, pcaTest, yTest)

    # plot the ROC part
    colors = ['aqua', 'darkorange']
    labelList = ['none', 'PCA']
    plt.figure()
    for i, color in zip(range(3), colors):
        plt.plot(fpr[i], tpr[i], color=color,
                 lw=2,
                 label=labelList[i])
    plt.xlabel('False Positive Rate')
```

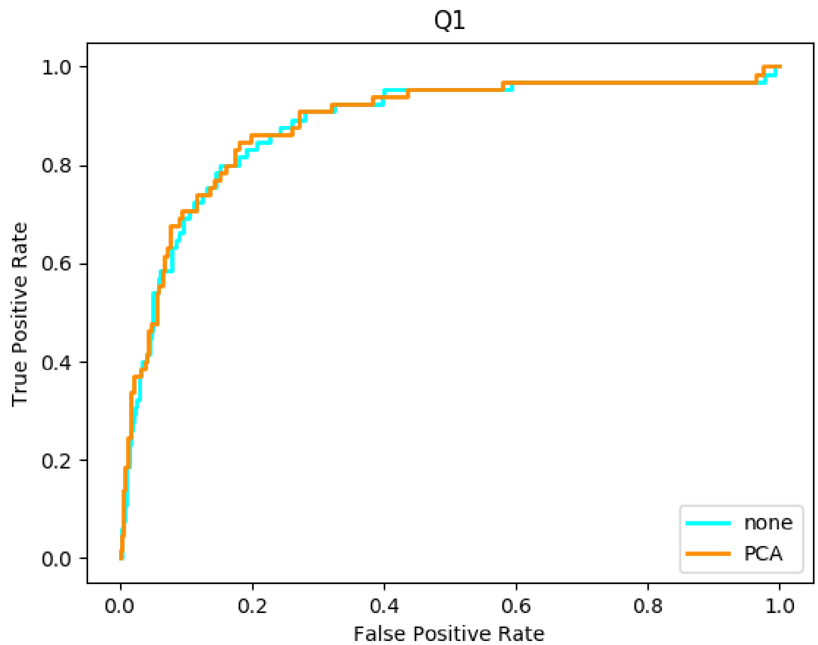

Figure 1: ROC curves

```
    plt.ylabel('True Positive Rate')
    plt.title('Q1')
    plt.legend(loc="lower right")
    plt.savefig('q1.png')


if __name__ == "__main__":
    main()
```

The resulting ROC curves are shown in Figure 1. From the plot, we can see that there is a range in which PCA outperforms the normalized dataset. This suggests that we can reduce a little bit of the features while still retaining predictive power.

2. **(30+15+5=45 pts) Almost Random Forest** The full code can be found in `rf.py` and `q2.py`.

   (a) The key parts are the train and predict function and doing book-keeping. For the solutions, each tree is stored in a dictionary, to keep track of the features used for that tree (this is necessary for prediction).

```
    def train(self, xFeat, y):
        """
        Train the random forest using the data

        Parameters
        ----------
        xFeat : nd-array with shape n x d
            Training data
        y : 1d array with shape n
            Array of responses associated with training data.
```

```python
        Returns
        -------
        stats : object
            Keys represent the number of trees and
            the values are the out of bag errors
        """
        n = xFeat.shape[0]
        d = xFeat.shape[1]
        nArr = range(n)
        oobPredict = collections.defaultdict(list)
        oobErr = {}
        # iterate through the m trees
        for m in range(self.nest):
            # bootstrap indices
            trainIdx = np.random.choice(nArr, n, replace=True)
            # get those that are oob
            oobIdx = np.setdiff1d(nArr, trainIdx)
            # get the feature idx
            featIdx = np.random.choice(d, self.maxFeat, replace=False)
            # subset the bootstrap data
            xTemp = xFeat[trainIdx][:, featIdx]
            yTemp = y[trainIdx]
            # train the model and store the stuff
            self.model[m] = {"tree": skt.DecisionTreeClassifier(criterion=self.criterion,
                                        max_depth=self.maxDepth,
                                        min_samples_leaf=self.minLeafSample),
                             "feat": featIdx,
                             "oob": oobIdx}
            self.model[m]["tree"].fit(xTemp, yTemp)
            # predict the OOB
            xOob = xFeat[oobIdx][:, featIdx]
            yOob = self.model[m]["tree"].predict(xOob)
            for k, idx in enumerate(oobIdx):
                oobPredict[idx].append(yOob[k])
            # predict all OOB
            oobHat = {k: ((np.sum(v)/ len(v)) >= 0.5)*1 for k, v in oobPredict.items()}
            # calculate errors
            oobErr[m] = 1-skm.accuracy_score(y[list(oobHat.keys())],
                                             list(oobHat.values()))
        return oobErr

    def predict(self, xFeat):
        """
        Given the feature set xFeat, predict
        what class the values will have.

        Parameters
        ----------
        xFeat : nd-array with shape m x d
            The data to predict.

        Returns
        -------
        yHat : 1d array or list with shape m
            Predicted response per sample
        """
        yHat = np.zeros(xFeat.shape[0])
        for m in range(self.nest):
            xTemp = xFeat[:, self.model[m]["feat"]]
            yHat = yHat + self.model[m]["tree"].predict(xTemp)
        yHat = (yHat > m/2) * 1
        return yHat
```

(b) We can use OOB error to determine the best parameter. For the solutions, we try 5-12 features, 1-6 max depth, and 5-8 maximum leaf samples (note that from HW2, we found that the optimal depth was 3 and leaf sample was 5).

```python
def search_param(xTrain, yTrain, xTest, yTest):
    perf = pd.DataFrame()
    # do a grid search using OOB and set max trees to 250
    for crit in ['gini', 'entropy']:
        for mf in range(5,12):
            for md in range(1,6):
                for mls in range(5,8):
                    print(crit, mf, md, mls)
                    rf = RandomForest(250, mf,
                                      crit, md,
                                      mls)
                    oobErr = rf.train(xTrain, yTrain)
                    # use oob to keep track of stuff
                    tmpDF = pd.DataFrame.from_dict(oobErr,
                                                   orient='index',
                                                   columns=['err'])
                    tmpDF['nest'] = tmpDF.index
                    tmpDF['crit'] = crit
                    tmpDF['mf'] = mf
                    tmpDF['md'] = md
                    tmpDF['mls'] = mls
                    perf = pd.concat([perf, tmpDF])
    # clean up the indexing for the pandas dataframe
    perf = perf.reset_index(drop=True)
    return perf
```

If we choose the first instance of the lowest OOB error, we get the following summary table:

err	nest	crit	mf	md	mls
0.109026	72	gini	11	5	7
0.109026	77	gini	8	5	5
0.109026	117	entropy	9	4	7
0.109026	47	gini	7	4	5
0.109026	188	gini	6	4	5
0.109920	91	entropy	8	4	6
0.109920	127	entropy	8	5	5
0.109920	117	entropy	8	5	6
0.109920	63	gini	9	4	6
0.109920	15	gini	8	5	7
0.109920	62	entropy	10	4	6
0.109920	202	entropy	7	5	7
0.109920	28	gini	6	5	7

Note that since there are multiple values with the same lowest error, it's better to chose the 'simpler one', which involves less depth and less number of trees. As a result, the optimal is 47 trees, 7 maximum features, 4 maximum depth, and 5 maximum leafs per sample.

(c) If we train using the optimal parameters from above, we get the following lines of code:

```python
def eval_opt(xTrain, yTrain, xTest, yTest):
    bst = RandomForest(47, 7, 'gini', 4, 5)
    ypred = bst.predict(xTest)
    # evaluate predictions
    print(1-skm.accuracy_score(yTest, ypred))
```

The error is 0.099999 on the test set, which is slightly below what OOB error predicts but very similar.