CS 334: Homework #2 Solutions

1. **Decision Tree**: The full code can be viewed in `dt.py`

   (a) At a high-level, the idea is to use recursion and adopt a data structure similar to the binary search tree. Under the recursion paradigm, there are 3 steps that are necessary to building the tree (1) determine the stopping condition (when it is a leaf), (2) determine the best splitting criteria and then (3) recursively build the left and right tree (so it'll look something like this):

```
    def decision_tree(self, xFeat, y, depth):
        # Stopping criteria
        if self.is_leaf(xFeat, y, depth):
            return stats.mode(y)[0]
        # find the split
        splitVar, splitVal = find_split(xFeat, y,
                                        self.criterion,
                                        self.minLeafSample)
        # Partition data into two sets
        xL, xR, yL, yR = partition_data(xFeat, y,
                                        splitVar, splitVal)
        # Recursive call of decision_tree()
        return {"split_variable": splitVar,
                "split_value": splitVal,
                "right": self.decision_tree(xR,
                                            yR,
                                            depth+1),
                "left": self.decision_tree(xL,
                                           yL,
                                           depth+1)}
```

   The splitting criteria requires checking through each feature and the possible splits, and storing the previous values:

```
def find_split(xFeat, y, criterion, minLeafSample):
    # go through each feature while keeping track of the best scenario
    # initialize them to the first column + value
    bestSplitVar = xFeat.columns[0]
    bestSplitVal = xFeat.iloc[0, 0]
    bestScore = 1    # set the score to be 1
    for splitVar in xFeat.columns:
        # sort the feature based on the column
        idx = np.argsort(xFeat[splitVar])
        xSorted = xFeat.iloc[idx, :]
        ySorted = y[idx]
        # check all possible splits starting at the min sample
        for i in range(minLeafSample, len(xFeat)-minLeafSample):
            splitVal = xSorted[splitVar].iloc[i]
            # check if the next value is the same as this one
            # if so then just check the next one anyways
            if ySorted[i] == ySorted[i+1]:
                continue
            _, _, yL, yR = partition_data(xSorted,
                                          ySorted,
                                          splitVar,
                                          splitVal)
            sL = calculate_split_score(yL, criterion)
            sR = calculate_split_score(yR, criterion)
            score = float(len(yL) / len(y)) * sL + float(len(yR) / len(y)) * sR
            if score < bestScore:
                # update the split
                bestSplitVar = splitVar
                bestSplitVal = splitVal
```

```
                bestScore = score
    return bestSplitVar, bestSplitVal
```

(b) The predict function involves walking through the tree until you hit a leaf (denoted as something that is not a `dict`) in our implementation.

```
def predict_sample(self, node, x):
    newNode = node['left']
    if x[node['split_variable']] > node['split_value']:
        newNode = node['right']
    # check if it's a dictionary which is another node
    if isinstance(newNode, dict):
        # recursive call of predict_sample
        return self.predict_sample(newNode, x)
    else:
        return newNode
```

(c) Create a 3D plot using following code.

```
maxDepth = np.arange(1,10)
minLeaf = np.arange(1,10)
trainauc = np.ones((9,9))
testauc = np.ones((9,9))
X, Y = np.meshgrid(maxDepth, minLeaf)

for m in maxDepth:
    for l in minLeaf:
        dt = DecisionTree('gini', m, l)
        trainauc[m-1,l-1], testauc[m-1, l-1] = dt_train_test(dt, xTrain, yTrain, xTest, yTest)
fig = plt.figure(figsize=(9,5))
ax = plt.axes(projection="3d")
ax.plot_wireframe(X, Y, trainauc, color='g',label='Train')
ax.plot_wireframe(X, Y, testauc, color='r',label='Test')
ax.set_title('3D plot of accuracy using Gini')
ax.set_xlabel('Min Leaf Samples')
ax.set_ylabel('Tree Depth')
ax.set_zlabel('Accuracy')
ax.legend()
plt.savefig('q1c.eps', format='eps', dpi=1000)
plt.show()
```

Figure 1 is a sample plot.

Create 2d plots with fixed values using the following code. Figure 2 is a sample output of 2D plots.

```
    maxDepth = list(np.arange(1,10))
    minLeaf = list(np.arange(1,10))
    yDepth = np.ones((9,2))
    yLeaf = np.ones((9,2))

    for m in maxDepth:
            dt = DecisionTree('gini', m, 5)
            trainauc, testauc = dt_train_test(dt, xTrain, yTrain, xTest, yTest)
            yDepth[m-1,0] = trainauc
            yDepth[m-1,1] = testauc

    for l in minLeaf:
            dt = DecisionTree('gini', 5, l)
            trainauc, testauc = dt_train_test(dt, xTrain, yTrain, xTest, yTest)
            yLeaf[l-1,0] = trainauc
            yLeaf[l-1,1] = testauc


    plt.subplot(1,2,1)
```

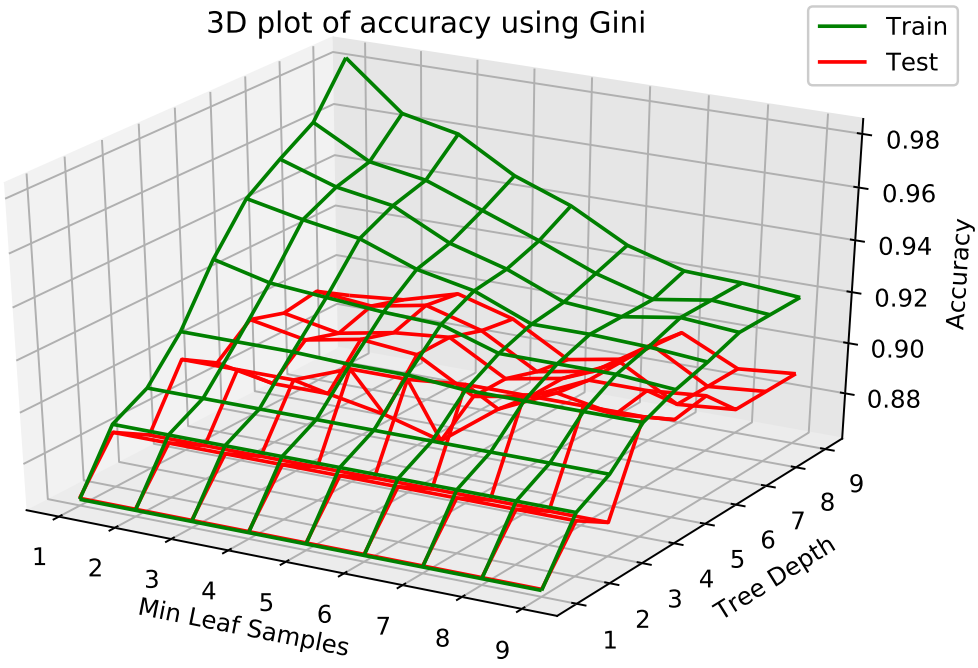

Figure 1: Q1c: Plot of accuracy of train and test as a function of parameters

```
plt.plot(maxDepth, yDepth[:,0], color = 'g', label='Train')
plt.plot(maxDepth, yDepth[:,1], color = 'r', label='Test')
plt.xlabel('Tree Depth')
plt.ylabel('Accuracy')
plt.legend(loc="upper left")

plt.subplot(1,2,2)
plt.plot(minLeaf, yLeaf[:,0], color = 'g', label='Train')
plt.plot(minLeaf, yLeaf[:,1], color = 'r', label='Test')
plt.xlabel('Min Leaf Samples')

plt.suptitle('2D plots of accuracy using gini')
plt.savefig('q1c_2d.png', format='png', dpi=1000)
plt.show()
```

(d) The computational cost of the predicting a single sample is just traversing the tree which is $O(p)$. The training cost is much more expensive as you need to test the splits at each point, and look at all the features. Assuming the worst case, where you test all the possible splits n for each feature, then the cost of each feature is to sort the feature + compute the scoring n times $(n \log n + n)$. Thus each node, the computation is $d(n \log n + n)$, and thus expect the complexity of train to be $\mathrm{O}(pdn \log n)$.

2. **Exploring Model Assessment Strategies**: The full code can be viewed in `q2.py`

(a) Under the model_selection module in `scikit-learn`, the holdout can be implemented using the `train_test_split` function.

```
def holdout(model, xFeat, y, testSize):
    """
    Split xFeat into random train and test based on the testSize

    Parameters
```

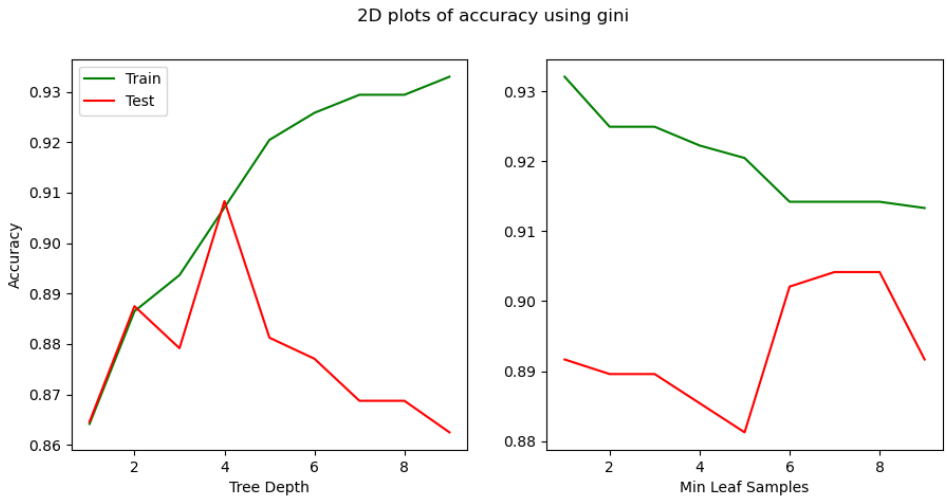

Figure 2: Q1c: 2D plots of accuracy of train and test as a function of parameters

```
    ----------
    model : sktree.DecisionTreeClassifier
        Decision tree model
    xFeat : nd-array with shape n x d
        Features of the dataset
    y : 1-array with shape n x 1
        Labels of the dataset
    testSize : float
        Portion of the dataset to serve as a holdout.

    Returns
    -------
    trainAuc : float
        Average AUC of the model on the training dataset
    testAuc : float
        Average AUC of the model on the validation dataset
    """
    trainAuc = 0
    testAuc = 0
    timeElapsed = 0
    start = time.time()
    xTrain, xTest, yTrain, yTest = skms.train_test_split(xFeat,
                                                         y,
                                                         test_size=testSize)
    trainAuc, testAuc = sktree_train_test(model,
                                          xTrain,
                                          yTrain,
                                          xTest,
                                          yTest)
    timeElapsed = time.time() - start
    return trainAuc, testAuc, timeElapsed
```

(b) Under the model_selection module in `scikit-learn`, the k-fold CV can be implemented using the `KFold` class where you create a new KFold object with the right number of splits.

```
def kfold_cv(model, xFeat, y, k):
    """
    Split xFeat into k different groups, and then use each of the
    k-folds as a validation set, with the model fitting on the remaining
    k-1 folds.
```

```
    Parameters
    ----------
    model : sktree.DecisionTreeClassifier
        Decision tree model
    xFeat : nd-array with shape n x d
        Features of the dataset
    y : 1-array with shape n x 1
        Labels of the dataset
    k : int
        Number of folds or groups (approximately equal size)

    Returns
    -------
    trainAuc : float
        Average AUC of the model on the training dataset
    testAuc : float
        Average AUC of the model on the validation dataset
    """
    # TODO FILL IN
    trainAuc = 0
    testAuc = 0
    timeElapsed = 0
    start = time.time()
    kf = skms.KFold(n_splits=k)
    for trainIdx, testIdx in kf.split(xFeat):
        xTrain, xTest = xFeat.iloc[trainIdx], xFeat.iloc[testIdx]
        yTrain, yTest = y.iloc[trainIdx], y.iloc[testIdx]
        temp1, temp2 = sktree_train_test(model,
                                         xTrain,
                                         yTrain,
                                         xTest,
                                         yTest)
        trainAuc = trainAuc + temp1
        testAuc = testAuc + temp2
    timeElapsed = time.time() - start
    return trainAuc / k, testAuc / k, timeElapsed
```

(c) Using the implementation for (a), Monte Carlo CV repeats the holdout s times and averages the AUC from the different samples.

```
def mc_cv(model, xFeat, y, testSize, s):
    """
    Perform s-samples of the Monte Carlo cross validation approach where
    for each sample you split xFeat into random train and test based on the testSize

    Parameters
    ----------
    model : sktree.DecisionTreeClassifier
        Decision tree model
    xFeat : nd-array with shape n x d
        Features of the dataset
    y : 1-array with shape n x 1
        Labels of the dataset
    testSize : float
        Portion of the dataset to serve as a holdout.

    Returns
    -------
    trainAuc : float
        Average AUC of the model on the training dataset
    testAuc : float
        Average AUC of the model on the validation dataset
    """
    # TODO FILL IN
    trainAuc = 0
    testAuc = 0
```

```
    timeElapsed = 0
    start = time.time()
    for sample in range(s):
        temp1, temp2, temp3 = holdout(model, xFeat, y, testSize)
        trainAuc = trainAuc + temp1
        testAuc = testAuc + temp2
        timeElapsed = timeElapsed + temp3
    timeElapsed = time.time() - start
    return trainAuc / s, testAuc / s, timeElapsed
```

(d) While the output of the script will vary each time you run it (unless you set a seed at the beginning), here is a sample output. Note that we had to run it twice to get accurate numbers as the holdout time will take longer since the dataset has not been loaded in the cache.

	Strategy	TrainAUC	ValAUC	Time
0	Holdout	0.950624	0.776385	0.006821
1	2-fold	0.955713	0.774866	0.010512
2	5-fold	0.952979	0.799876	0.034826
3	10-fold	0.954147	0.793916	0.072114
4	MCCV w/ 5	0.950379	0.806608	0.034564
5	MCCV w/ 10	0.952376	0.800440	0.068880
6	True Test	0.952502	0.816793	0.000000

The important column to look at is the validation AUC and the time. What we can see is the MCCV and k-fold CV can get a more robust estimate of the validation AUC. However, the time it takes is longer due to the book-keeping nature of k-fold CV compared to MCCV.

3. **Robustness of Decision Trees and K-NN**: The full code can be viewed in `q3.py`

(a) For the purpose of this solution, we'll choose $k = 5$ since the dataset is relatively small and we want to maximize the training data without requiring too much computational time. In addition, the results from question2 suggests $k = 5$ had a pretty accurate estimate of the test AUC. We will also use `GridSearchCV` from `scikit-learn` to calculate the best parameter using the best AUC score. Note that k-nn and decision tree will share similar code (other than the model), so we created a function to find the optimal parameters and test the robustness of the model. Based on the 5-fold, the optimal parameter for knn is 11 neighbors.

```
def model_opt(model, modelParams,
              xTrain, yTrain,
              xTest, yTest):
    gs = skms.GridSearchCV(model,
                           modelParams,
                           cv=5,
                           scoring='roc_auc')
    gsResult = gs.fit(xTrain, yTrain)
    print(gsResult.best_params_)
    nTrain = len(xTrain)
    robustStats = {}
    for rRate in [0, 0.01, 0.05, 0.1]:
        # take a subset of the samples
        nNew = int(nTrain*(1-rRate))
        newIdx = np.random.choice(nTrain, nNew, replace=False)
        xTrainNew = xTrain.iloc[newIdx, :]
        yTrainNew = yTrain[newIdx]
        # train and test
```

```
        perf = train_test(gsResult.best_estimator_,
                          xTrainNew, yTrainNew,
                          xTest, yTest)
        robustStats[rRate] = perf
    return pd.DataFrame(robustStats)
```

(b) While the numbers will change each time for the run:

	0.00	0.05	0.10	0.20
trainAuc	0.851946	0.848214	0.848340	0.840650
testAuc	0.790621	0.788916	0.754476	0.755514
trainAcc	0.865952	0.864534	0.869911	0.869274
testAcc	0.862500	0.866667	0.864583	0.866667

(c) For the decision tree, the optimal parameters are the Gini criterion, max depth of 3, and minimum number of samples is 5. The numbers that we got from removing random samples:

	0.00	0.05	0.10	0.20
trainAuc	0.840816	0.834958	0.858418	0.842576
testAuc	0.848860	0.872994	0.842595	0.862892
trainAcc	0.893655	0.892756	0.887786	0.886034
testAcc	0.879167	0.879167	0.877083	0.879167

(d) The output from (b) and (c) is summarized in Table 1. What we observe is that in the case of both models, is that the test accuracy fluctuates quite a bit based on the training data. In particular, there is a drop in the test AUC the more training data is removed. However, accuracy seems to hold fairly steady (somewhat surprisingly).

Model	Missing	Train AUC	Test AUC	Train Acc	Test Acc
K-nn	0.00	0.851946	0.790621	0.865952	0.862500
K-nn	0.05	0.848214	0.788916	0.864534	0.866667
K-nn	0.10	0.848340	0.754476	0.869911	0.870833
K-nn	0.20	0.840650	0.755514	0.869274	0.866667
Decision Tree	0.00	0.840816	0.848860	0.893655	0.879167
Decision Tree	0.05	0.834958	0.872994	0.892756	0.879167
Decision Tree	0.10	0.858418	0.842595	0.887786	0.877083
Decision Tree	0.20	0.842576	0.862892	0.886034	0.879167

Table 1: Robustness of the two models