This assignment represents my own work. I did not work on this assignment with others. All coding was done by myself.

I understand that if I struggle with this assignment that I will reevaluate whether this is the correct class for me to take. I understand that the homework only gets harder.

CS 671: Homework 1

Alex Kumar

Question 5.

In [ ]:
```
# Imports
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from chefboost import Chefboost as chef
```

In [ ]:
```
### General Data Helper Functions

# Returns train and test data
def getData():
    return pd.read_csv("carseats_train.csv"), pd.read_csv("carseats_test.csv")

# Separates the feature vector from the label associated with it
def splitXY(data):
    return data.iloc[:, 1:].to_numpy(), data.iloc[:, 0].to_numpy()

# Change categorical data to numerical
def cleanX(data):
    old, new = ["Bad", "Medium", "Good", "No", "Yes"], [0, 1, 2, 0, 1]
    for i in range(len(old)):
        data[data == old[i]] = new[i]
    return data

# Renames and relocates the label column; changes numerical to categorical
def chefPrep(data):
    data = data.rename(columns={"Sales": "Decision"})
    dec = data.pop("Decision")
    data.insert(len(data.columns), "Decision", dec)

    data.loc[data["Decision"] == 0, "Decision"] = "No"
    data.loc[data["Decision"] == 1, "Decision"] = "Yes"
    return data


### For CV
# Split data into K folds as np array for CV
def kFolds(data_X, data_Y, K=5):
    folds_X, folds_Y = [], []
    bucket = len(data_X) // K
```

```
    start, end = 0, bucket
    for i in range(K):
        if i+1 == K:
            folds_X.append(data_X[start:])
            folds_Y.append(data_Y[start:])
        else:
            folds_X.append(data_X[start: end])
            folds_Y.append(data_Y[start: end])
        start += bucket
        end += bucket
    return folds_X, folds_Y

# Prepare train / valid pairs for CV
def cvSplit(data_X, data_Y, K=5):
    validX, validY, trainX, trainY = [], [], [], []
    folds = list(range(K))
    for i in range(K):
        vX, vY, tX, tY = [], [], [], []
        temp = folds.copy()
        temp.remove(i)
        vX, vY = data_X[i], data_Y[i]
        for j in temp:
            if len(tX) == 0:
                tX, tY = data_X[j], data_Y[j]
            else:
                tX = np.concatenate((data_X[j], tX))
                tY = np.concatenate((data_Y[j], tY))
        validX.append(vX)
        validY.append(vY)
        trainX.append(tX)
        trainY.append(tY)
    return validX, validY, trainX, trainY

# Split data into K folds as pd dataframe for CV
def kFoldsChef(data, K):
    folds = []
    bucket = len(data) // K
    start, end = 0, bucket
    for i in range(K):
        if i+1 == K:
            folds.append(data.iloc[start:])
        else:
            folds.append(data.iloc[start: end])
        start += bucket
        end += bucket
    return folds

# Prepare train / valid pairs for CV
def cvSplitChef(data, K):
    valid, train = [], []
    folds = list(range(K))
    for i in range(K):
        v, t = [], []
        temp = folds.copy()
        temp.remove(i)
        v = data[i]
        for j in temp:
```

```
            if len(t) == 0:
                t = data[j]
            else:
                t = pd.concat([data[j], t])
        valid.append(v)
        train.append(t)
    return valid, train
```

In [ ]:
```
### Evalutation Helper Functions

# Calculate confusion matrix for 1/0
def confusionMatrix(preds, labels):
    confusion = [0, 0, 0, 0]    # [TP, FP, FN, TN]
    for i in range(len(preds)):
        p, l = preds[i], labels[i]
        if p == 1:                                  # pred pos
            if l == 1: confusion[0] += 1            # TP
            else: confusion[1] += 1                 # FP
        else:                                       # pred neg
            if l == 0: confusion[3] += 1            # TN
            else: confusion[2] += 1                 # FN
    return confusion

# Calculate confusion matrix for Yes/No
def chefConfusion(preds, labels):
    confusion = [0, 0, 0, 0]    # [TP, FP, FN, TN]
    for i in range(len(preds)):
        p, l = preds[i], labels[i]
        if p == "Yes":                              # pred pos
            if l == "Yes": confusion[0] += 1        # TP
            else: confusion[1] += 1                 # FP
        else:                                       # pred neg
            if l == "No": confusion[3] += 1         # TN
            else: confusion[2] += 1                 # FN
    return confusion

# Precision
def calcPrecision(c):
    return c[0] / (c[0] + c[1])     # [TP, FP, FN, TN]

# Recall
def calcRecall(c):
    return c[0] / (c[0] + c[2])

# F1
def calcF1(c):
    p = calcPrecision(c)
    r = calcRecall(c)
    return 2 * ((p * r) / (p + r))
```

In [ ]:
```
# KNN Helper Functions

# Returns euclidean distance
def euclidDist(x, y):
    dist = []
    for i in range(len(x)):
        dist.append((x[i] - y[i]) ** 2)
```

```
    return np.sqrt(sum(dist))

# Returns manhattan distance
def manhatDist(x, y):
    dist = []
    for i in range(len(x)):
        dist.append(np.abs(x[i] - y[i]))
    return sum(dist)

# Finds distance from p to all points in train, returns closest k
def distToTrain(train_X, train_Y, p, dist_measure, k):
    distances = []
    for i in range(len(train_X)):
        if dist_measure == "euclidean":
            dist = euclidDist(train_X[i], p)
        elif dist_measure == "manhattan":
            dist = manhatDist(train_X[i], p)
        distances.append((dist, train_Y[i]))
    sorted_dist = sorted(distances)
    return sorted_dist[:k]

# Returns majority label from closest k
def getMajority(distances):
    g1, g2 = 0, 0
    for d in distances:
        if d[1] == 0: g1 += 1
        elif d[1] == 1: g2 += 1
    return 0 if g1 > g2 else 1
```

In [ ]:
```
# Cross Validation Helper Functions

######### COMBINE INTO 1 FUNCT WITH CALL TO FUNCT TO RUN MODEL
# Get summed f1 scores across K folds for each hyper for Decision Tree
def rotateCV(trainX, trainY, validX, validY, hyper, K):
    fold_vals = {}
    for i in range(K):
        for h in hyper:
            cv_tree = DecisionTreeClassifier(max_depth=h, random_state=None)
            cv_tree = cv_tree.fit(trainX[i], trainY[i])

            cv_predict = cv_tree.predict(validX[i])
            confusion_cv = confusionMatrix(cv_predict, validY[i])
            f1_cv = calcF1(confusion_cv)
            # print("{a}th fold f1 score for h value {b}: ".format(a=i, b=h), f
            if h not in fold_vals.keys():
                fold_vals[h] = f1_cv
            else:
                fold_vals[h] += f1_cv
    return fold_vals

# Get summed f1 scores across K folds for each hyper for chefboost
def rotateCVChef(train, valid, hyper, K):
    fold_vals = {}
    for i in range(K):
        for h in hyper:
            config = {"algorithm": "CART", "max_depth": h}
            chef_cv = chef.fit(train[i], config=config)
```

```
            chef_predict = []
            for j in range(len(valid[i])):
                chef_predict.append(chef.predict(chef_cv, valid[i].iloc[j]))
            confusion = chefConfusion(chef_predict, valid[i]["Decision"].to_lis
            f1_cv = calcF1(confusion)

            if h not in fold_vals.keys():
                fold_vals[h] = f1_cv
            else:
                fold_vals[h] += f1_cv
    return fold_vals

# Find hyperparam with best average score across K folds
def findBestK(fold_vals, K):
    for k in fold_vals.keys():
        fold_vals[k] = fold_vals[k] / K

    return max(fold_vals, key=fold_vals.get)

# Gets F1 score for current setting of hyperparams on valid set
def KNNCV(train_X, train_Y, valid_X, valid_Y, k, dist_meas):
    preds = []
    for i in range(len(valid_X)):
        dist = distToTrain(train_X, train_Y, valid_X[i], dist_meas, k)
        vote = getMajority(dist)
        preds.append(vote)

    knn_confusion = confusionMatrix(preds, valid_Y)
    return calcF1(knn_confusion)
```

In [ ]:

```
# Decision Tree Main Function
def decTree():
    # Get and clean the data
    train, test = getData()
    train_X, train_Y = splitXY(train)
    test_X, test_Y = splitXY(test)
    train_X = cleanX(train_X)
    test_X = cleanX(test_X)

    # Train the model
    decision_tree = DecisionTreeClassifier(max_depth=3, random_state=None)
    decision_tree = decision_tree.fit(train_X, train_Y)

    # Get test F1 score
    dt_predict = decision_tree.predict(test_X)
    confusion = confusionMatrix(dt_predict, test_Y)
    f1 = calcF1(confusion)
    # print("Decision Tree F1 Score: ", f1)

    # CV
    K, hyper = 5, [1, 2, 3, 4]
    folds_X, folds_Y = kFolds(train_X, train_Y, K)
    # Split into sets that are rotated
    validX, validY, trainX, trainY = cvSplit(folds_X, folds_Y, K)

    # Train over all h
    fold_vals = rotateCV(trainX, trainY, validX, validY, hyper, K)
```

```
    # Find best h
    max_key = findBestK(fold_vals, K)
    print("Best value of hyperparmeter being tuned: ", max_key)

    # Make model with optimal hyperparam
    optimal_tree = DecisionTreeClassifier(max_depth=max_key, random_state=None)
    optimal_tree = optimal_tree.fit(train_X, train_Y)

    # Get F1
    optimal_predict = optimal_tree.predict(test_X)
    confusion_optimal = confusionMatrix(optimal_predict, test_Y)
    f1_optimal = calcF1(confusion_optimal)
    print("Normal Decision Tree F1 Score: ", f1)
    print("F1 score after CV tuning: ", f1_optimal)
    return
```

In [ ]:

```
# Chefboost Main Function
def chefTime():
    # Get and clean the data
    train, test = getData()
    train = chefPrep(train)
    test = chefPrep(test)

    # Train the model
    config = {"algorithm": "CART", "max_depth": 5}
    chef_model = chef.fit(train, config=config)

    # Get test F1 Score
    chef_predict = []
    for i in range(len(test)):
        chef_predict.append(chef.predict(chef_model, test.iloc[i]))
    confusion = chefConfusion(chef_predict, test["Decision"].to_list())
    f1 = calcF1(confusion)
    # print("Chef F1 Score: ", f1)

    # CV
    K, hyper = 5, [1, 2, 3, 4]
    folds = kFoldsChef(train, K)
    # Split into sets that are rotated
    valid_cv, train_cv = cvSplitChef(folds, K)
    # Train over all h
    fold_vals = rotateCVChef(train_cv, valid_cv, hyper, K)

    # Find best h
    max_key = findBestK(fold_vals, K)
    print("Best value of hyperparmeter being tuned: ", max_key)

    # Make model with optimal hyperparam
    config = {"algorithm": "CART", "max_depth": max_key}
    chef_optimal = chef.fit(train, config=config)

    # Get F1
    optimal_predict = []
    for i in range(len(test)):
        optimal_predict.append(chef.predict(chef_optimal, test.iloc[i]))
    confusion_opt = chefConfusion(optimal_predict, test["Decision"].to_list())
```

```
    f1_optimal = calcF1(confusion_opt)
    print("Normal Chef F1 Score: ", f1)
    print("Chef F1 Score after CV Tuning: ", f1_optimal)
    return
```

In [ ]:

```
# KNN Main Function
def KNN():
    # Get and clean data
    train, test = getData()
    train_X, train_Y = splitXY(train)
    test_X, test_Y = splitXY(test)
    train_X = cleanX(train_X)
    test_X = cleanX(test_X)

    # Run KNN CV
    dist_measures = ["euclidean", "manhattan"]
    num_neighbors = [1, 3, 5, 7, 9]
    K = 5
    folds_X, folds_Y = kFolds(train_X, train_Y, K)
    # Split into sets that are rotated
    validX, validY, trainX, trainY = cvSplit(folds_X, folds_Y, K)

    fold_vals = {}
    for d in dist_measures:
        for n in num_neighbors:
            for i in range(len(trainX)):
                cv_f1 = KNNCV(trainX[i], trainY[i], validX[i], validY[i], n, d)
                if (d,n) not in fold_vals.keys():
                    fold_vals[(d,n)] = cv_f1
                else:
                    fold_vals[(d,n)] += cv_f1
                # print("Dist {d}; NumNeih {n}; Fold {i} F1 score: ".format(d=c

    max_key = findBestK(fold_vals, K)
    print("Best distance and NN parameters: ", max_key)

    # Performance of Optimal KNN
    preds = []
    for i in range(len(test_X)):
        dist = distToTrain(train_X, train_Y, test_X[i], max_key[0], max_key[1])
        vote = getMajority(dist)
        preds.append(vote)

    knn_confusion = confusionMatrix(preds, test_Y)
    knn_f1 = calcF1(knn_confusion)
    print("KNN F1 score after CV tuning: ", knn_f1)
    return
```

In [ ]:

```
# Run Decision Tree Function
decTree()
```

```
Best value of hyperparmeter being tuned:  4
Normal Decision Tree F1 Score:  0.5777777777777777
F1 score after CV tuning:  0.6923076923076924
```

In [ ]:

```
# Run Chefboost Function
chefTime()
```

```
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.641408920288086  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  97.16312056737588 % on  282  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[100, 3], [5, 174]]
Precision:  97.0874 %, Recall:  95.2381 %, F1:  96.1539 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.431269884109497  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  98.67256637168141 % on  226  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[82, 0], [3, 141]]
Precision:  100.0 %, Recall:  96.4706 %, F1:  98.2036 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.3990559577941895  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  98.67256637168141 % on  226  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[82, 0], [3, 141]]
Precision:  100.0 %, Recall:  96.4706 %, F1:  98.2036 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.5331168174743652  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  98.67256637168141 % on  226  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[82, 0], [3, 141]]
Precision:  100.0 %, Recall:  96.4706 %, F1:  98.2036 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.4706659317016602  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  98.67256637168141 % on  226  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[82, 0], [3, 141]]
Precision:  100.0 %, Recall:  96.4706 %, F1:  98.2036 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
```

```
-------------------------
finished in  1.582603931427002  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  96.90265486725664 % on  226  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[79, 2], [5, 140]]
Precision:  97.5309 %, Recall:  94.0476 %, F1:  95.7576 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.5586268901824951  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  96.90265486725664 % on  226  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[79, 2], [5, 140]]
Precision:  97.5309 %, Recall:  94.0476 %, F1:  95.7576 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.6172471046447754  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  96.90265486725664 % on  226  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[79, 2], [5, 140]]
Precision:  97.5309 %, Recall:  94.0476 %, F1:  95.7576 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.5485520362854004  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  96.90265486725664 % on  226  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[79, 2], [5, 140]]
Precision:  97.5309 %, Recall:  94.0476 %, F1:  95.7576 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.577951192855835  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  98.67256637168141 % on  226  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[83, 1], [2, 140]]
Precision:  98.8095 %, Recall:  97.6471 %, F1:  98.2249 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.581491231918335  seconds
```

```
-------------------------
Evaluate  train set
-------------------------
Accuracy:  98.67256637168141 % on  226  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[83, 1], [2, 140]]
Precision:  98.8095 %, Recall:  97.6471 %, F1:  98.2249 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.581228256225586  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  98.67256637168141 % on  226  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[83, 1], [2, 140]]
Precision:  98.8095 %, Recall:  97.6471 %, F1:  98.2249 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.6014139652252197  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  98.67256637168141 % on  226  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[83, 1], [2, 140]]
Precision:  98.8095 %, Recall:  97.6471 %, F1:  98.2249 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.2408387660980225  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  96.01769911504425 % on  226  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[83, 5], [4, 134]]
Precision:  94.3182 %, Recall:  95.4023 %, F1:  94.8572 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.2702221870422363  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  96.01769911504425 % on  226  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[83, 5], [4, 134]]
Precision:  94.3182 %, Recall:  95.4023 %, F1:  94.8572 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.2755241394042969  seconds
-------------------------
Evaluate  train set
```

```
-------------------------
Accuracy:  96.01769911504425 % on  226  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[83, 5], [4, 134]]
Precision:  94.3182 %, Recall:  95.4023 %, F1:  94.8572 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.27650785446167  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  96.01769911504425 % on  226  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[83, 5], [4, 134]]
Precision:  94.3182 %, Recall:  95.4023 %, F1:  94.8572 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.4284229278564453  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  99.55357142857143 % on  224  instances
Labels:  ['No' 'Yes']
Confusion matrix:  [[145, 1], [0, 78]]
Precision:  99.3151 %, Recall:  100.0 %, F1:  99.6564 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.4346230030059814  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  99.55357142857143 % on  224  instances
Labels:  ['No' 'Yes']
Confusion matrix:  [[145, 1], [0, 78]]
Precision:  99.3151 %, Recall:  100.0 %, F1:  99.6564 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.4274659156799316  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  99.55357142857143 % on  224  instances
Labels:  ['No' 'Yes']
Confusion matrix:  [[145, 1], [0, 78]]
Precision:  99.3151 %, Recall:  100.0 %, F1:  99.6564 %
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.427408218383789  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  99.55357142857143 % on  224  instances
```

```
Labels:  ['No' 'Yes']
Confusion matrix:  [[145, 1], [0, 78]]
Precision:  99.3151 %, Recall:  100.0 %, F1:  99.6564 %
Best value of hyperparmeter being tuned:  1
[INFO]:  5 CPU cores will be allocated in parallel running
CART  tree is going to be built...
-------------------------
finished in  1.619767665863037  seconds
-------------------------
Evaluate  train set
-------------------------
Accuracy:  97.16312056737588 % on  282  instances
Labels:  ['Yes' 'No']
Confusion matrix:  [[100, 3], [5, 174]]
Precision:  97.0874 %, Recall:  95.2381 %, F1:  96.1539 %
Normal Chef F1 Score:  0.6434782608695652
Chef F1 Score after CV Tuning:  0.6434782608695652
```

In [ ]:

```
# Run KNN Function
KNN()
```

```
Best distance and NN parameters:  ('manhattan', 1)
KNN F1 score after CV tuning:  0.5535714285714286
```