final

May 17, 2023

[ ]:
```
import math
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from collections import defaultdict,Counter
from random import sample
from sklearn import datasets
from sklearn.model_selection import train_test_split
import NeuralNet as nn
import RandomForest as rf
```

0.1 General Notes

- This project is created by Andrei Treil and Sebastian McKay
- For all implementations, we used Andrei Treil's versions of the code, either by creating python files such as `NeuralNet.py` and `RandomForest.py` or by copying and pasting the code directly into cells

0.2 Hand-Written Digits Dataset

Models Used: - Neural Networks - KNN

0.2.1 1. Neural Network

1.1 discuss which algorithms you decided to test on each dataset and why > For the hand drawn numbers dataset, we decided to use neural networks due to the large size of the data set, as well as the data all being numeric.

[ ]:
```
digits = datasets.load_digits(as_frame=True)
dig_df = digits['data']
dig_df['class'] = digits['target']
dig_df.insert(0,'bias',1)

#split data by class into k groups the combine into folds
k = 10
dig_class_0 = dig_df.loc[dig_df['class'] == 0].sample(frac=1)
dig_class_0['class'] = [[1,0,0,0,0,0,0,0,0,0]] * len(dig_class_0)
dg0_split =  np.array_split(dig_class_0,k)
```

```
dig_class_1 = dig_df.loc[dig_df['class'] == 1].sample(frac=1)
dig_class_1['class'] = [[0,1,0,0,0,0,0,0,0,0]] * len(dig_class_1)
dg1_split =  np.array_split(dig_class_1,k)
dig_class_2 = dig_df.loc[dig_df['class'] == 2].sample(frac=1)
dig_class_2['class'] = [[0,0,1,0,0,0,0,0,0,0]] * len(dig_class_2)
dg2_split =  np.array_split(dig_class_2,k)
dig_class_3 = dig_df.loc[dig_df['class'] == 3].sample(frac=1)
dig_class_3['class'] = [[0,0,0,1,0,0,0,0,0,0]] * len(dig_class_3)
dg3_split =  np.array_split(dig_class_3,k)
dig_class_4 = dig_df.loc[dig_df['class'] == 4].sample(frac=1)
dig_class_4['class'] = [[0,0,0,0,1,0,0,0,0,0]] * len(dig_class_4)
dg4_split =  np.array_split(dig_class_4,k)
dig_class_5 = dig_df.loc[dig_df['class'] == 5].sample(frac=1)
dig_class_5['class'] = [[0,0,0,0,0,1,0,0,0,0]] * len(dig_class_5)
dg5_split =  np.array_split(dig_class_5,k)
dig_class_6 = dig_df.loc[dig_df['class'] == 6].sample(frac=1)
dig_class_6['class'] = [[0,0,0,0,0,0,1,0,0,0]] * len(dig_class_6)
dg6_split =  np.array_split(dig_class_6,k)
dig_class_7 = dig_df.loc[dig_df['class'] == 7].sample(frac=1)
dig_class_7['class'] = [[0,0,0,0,0,0,0,1,0,0]] * len(dig_class_7)
dg7_split =  np.array_split(dig_class_7,k)
dig_class_8 = dig_df.loc[dig_df['class'] == 8].sample(frac=1)
dig_class_8['class'] = [[0,0,0,0,0,0,0,0,1,0]] * len(dig_class_8)
dg8_split =  np.array_split(dig_class_8,k)
dig_class_9 = dig_df.loc[dig_df['class'] == 9].sample(frac=1)
dig_class_9['class'] = [[0,0,0,0,0,0,0,0,0,1]] * len(dig_class_9)
dg9_split =  np.array_split(dig_class_9,k)
dig_vals =␣
 ↪[[1,0,0,0,0,0,0,0,0,0],[0,1,0,0,0,0,0,0,0,0],[0,0,1,0,0,0,0,0,0,0],[0,0,0,1,0,0,0,0,0,0],[0

#list to hold folds
dig_fold = []
for i in range(k):
    this_fold =␣
 ↪[dg0_split[i],dg1_split[i],dg2_split[i],dg3_split[i],dg4_split[i],dg5_split[i],dg6_split[i]
    dig_fold.append(pd.concat(this_fold))

#dig_nn_arc =␣
 ↪[[64,64,10],[64,128,10],[64,64,128,10],[64,32,64,10],[64,64,32,64,10],[64,64,128,128,64,10]
dig_nn_arc = [[64,64,10],[64,128,10]]

def dig_test(fold,vals,nn_arc,lamb,eps,alpha,batch_size):
    dig_res = nn.k_fold(fold,vals,nn_arc,lamb,eps,alpha,batch_size)
    arc_dict = defaultdict(list)
    print(f'lamb = {lamb} eps = {eps} alpha = {alpha} batch_size =␣
 ↪{batch_size}')
```

```
    for arc,perf in dig_res.items():
        avg_acc,avg_f1 = [0,0]
        for res in perf:
            avg_acc += res[0]
            avg_f1 += res[1]
        arc_dict['Architecture'].append(arc)
        arc_dict['Accuracy'].append(avg_acc/10)
        arc_dict['F1'].append(avg_f1/10)

    arc_table = pd.DataFrame(arc_dict)
    print(arc_table)
```

1.3 To obtain the best performance possible, you should carefully adjust the hyper-parameters of each algorithm when deployed on a dataset

[ ]:
```
hyper_params = [[0.4,0.01,5,50],[0.6,0.01,5,50],[0.4,0.001,5,50],[0.6,0.
 ↪001,5,50]]
for params in hyper_params:
   ␣
 ↪dig_test(dig_fold,dig_vals,dig_nn_arc,params[0],params[1],params[2],params[3])
```

```
lamb = 0.4 eps = 0.01 alpha = 5 batch_size = 50
    Architecture  Accuracy        F1
0    [64, 64, 10]  0.955459  0.955166
1   [64, 128, 10]  0.958234  0.957735
lamb = 0.6 eps = 0.01 alpha = 5 batch_size = 50
    Architecture  Accuracy        F1
0    [64, 64, 10]  0.958786  0.959035
1   [64, 128, 10]  0.964924  0.964825
lamb = 0.4 eps = 0.001 alpha = 5 batch_size = 50
    Architecture  Accuracy        F1
0    [64, 64, 10]  0.964949  0.965058
1   [64, 128, 10]  0.967716  0.967639
lamb = 0.6 eps = 0.001 alpha = 5 batch_size = 50
    Architecture  Accuracy        F1
0    [64, 64, 10]  0.964918  0.964842
1   [64, 128, 10]  0.968833  0.968655
```

[ ]:
```
dig_nn_arc = [[64,64,10],[64,128,10],[64,64,128,10],[64,32,64,10]]
hyper_params = [[0.2,0.001,5,50],[0.4,0.001,5,50]]
for params in hyper_params:
   ␣
 ↪dig_test(dig_fold,dig_vals,dig_nn_arc,params[0],params[1],params[2],params[3])
```

```
lamb = 0.2 eps = 0.001 alpha = 5 batch_size = 50
        Architecture  Accuracy        F1
0       [64, 64, 10]  0.965455  0.965219
1      [64, 128, 10]  0.972150  0.972106
```

```
2  [64, 64, 128, 10]  0.962743  0.962414
3   [64, 32, 64, 10]  0.946058  0.945727
lamb = 0.4 eps = 0.001 alpha = 5 batch_size = 50
        Architecture  Accuracy        F1
0       [64, 64, 10]  0.957652  0.957369
1      [64, 128, 10]  0.972685  0.972667
2  [64, 64, 128, 10]  0.959403  0.959354
3   [64, 32, 64, 10]  0.945982  0.945710
```

[ ]:
```
dig_nn_arc = [[64,128,10],[64,128,128,10]]
hyper_params = [[0.2,0.0001,7,50],[0.2,0.0001,10,50]]
for params in hyper_params:
    dig_test(dig_fold,dig_vals,dig_nn_arc,params[0],params[1],params[2],params[3])
```

```
lamb = 0.2 eps = 0.0001 alpha = 7 batch_size = 50
         Architecture  Accuracy        F1
0       [64, 128, 10]  0.966064  0.965883
1  [64, 128, 128, 10]  0.972193  0.972465
lamb = 0.2 eps = 0.0001 alpha = 10 batch_size = 50
         Architecture  Accuracy        F1
0       [64, 128, 10]  0.966079  0.965760
1  [64, 128, 128, 10]  0.961644  0.961582
```

[ ]:
```
dig_nn_arc = [[64,64,10],[64,128,10],[64,128,128,10],[64,64,128,64,10]]
hyper_params = [[0.2,0.0001,5,50],[0.2,0.0001,7,50]]
for params in hyper_params:
    dig_test(dig_fold,dig_vals,dig_nn_arc,params[0],params[1],params[2],params[3])
```

```
lamb = 0.2 eps = 0.0001 alpha = 5 batch_size = 50
            Architecture  Accuracy        F1
0           [64, 64, 10]  0.959416  0.959145
1          [64, 128, 10]  0.969383  0.969181
2     [64, 128, 128, 10]  0.972754  0.972740
3  [64, 64, 128, 64, 10]  0.963294  0.963047
lamb = 0.2 eps = 0.0001 alpha = 7 batch_size = 50
            Architecture  Accuracy        F1
0           [64, 64, 10]  0.957667  0.957718
1          [64, 128, 10]  0.972792  0.972715
2     [64, 128, 128, 10]  0.970452  0.970365
3  [64, 64, 128, 64, 10]  0.962088  0.961845
```

[ ]:
```
dig_nn_arc = [[64,128,10],[64,128,128,10]]
hyper_params = [[0.4,0.001,5,50],[0.4,0.001,7,50]]
for params in hyper_params:
```

```
    ␣
  ↪dig_test(dig_fold,dig_vals,dig_nn_arc,params[0],params[1],params[2],params[3])
```

```
lamb = 0.4 eps = 0.001 alpha = 5 batch_size = 50
         Architecture  Accuracy        F1
0       [64, 128, 10]  0.972705  0.972602
1  [64, 128, 128, 10]  0.967140  0.967031
lamb = 0.4 eps = 0.001 alpha = 7 batch_size = 50
         Architecture  Accuracy        F1
0       [64, 128, 10]  0.960403  0.960171
1  [64, 128, 128, 10]  0.963249  0.963181
```

[ ]:
```
dig_nn_arc = [[64,128,10]]
hyper_params = [[0.1,0.0001,7,50],[0.05,0.0001,7,50]]
for params in hyper_params:
    ␣
  ↪dig_test(dig_fold,dig_vals,dig_nn_arc,params[0],params[1],params[2],params[3])
```

```
lamb = 0.1 eps = 0.0001 alpha = 7 batch_size = 50
    Architecture  Accuracy        F1
0  [64, 128, 10]  0.970007  0.969877
lamb = 0.05 eps = 0.0001 alpha = 7 batch_size = 50
    Architecture  Accuracy        F1
0  [64, 128, 10]   0.97151  0.971131
```

[ ]:
```
dig_nn_arc = [[64,128,10]]
hyper_params = [[0.1,0.0001,5,50],[0.05,0.0001,5,50]]
for params in hyper_params:
    ␣
  ↪dig_test(dig_fold,dig_vals,dig_nn_arc,params[0],params[1],params[2],params[3])
```

```
lamb = 0.1 eps = 0.0001 alpha = 5 batch_size = 50
    Architecture  Accuracy       F1
0  [64, 128, 10]  0.975509  0.97539
lamb = 0.05 eps = 0.0001 alpha = 5 batch_size = 50
    Architecture  Accuracy       F1
0  [64, 128, 10]  0.963858  0.96357
```

[ ]:
```
dig_nn_arc = [[64,128,10]]
hyper_params = [[0.1,0.0001,3,50],[0.1,0.0001,5,50]]
for params in hyper_params:
    ␣
  ↪dig_test(dig_fold,dig_vals,dig_nn_arc,params[0],params[1],params[2],params[3])
```

```
lamb = 0.1 eps = 0.0001 alpha = 3 batch_size = 50
    Architecture  Accuracy        F1
```

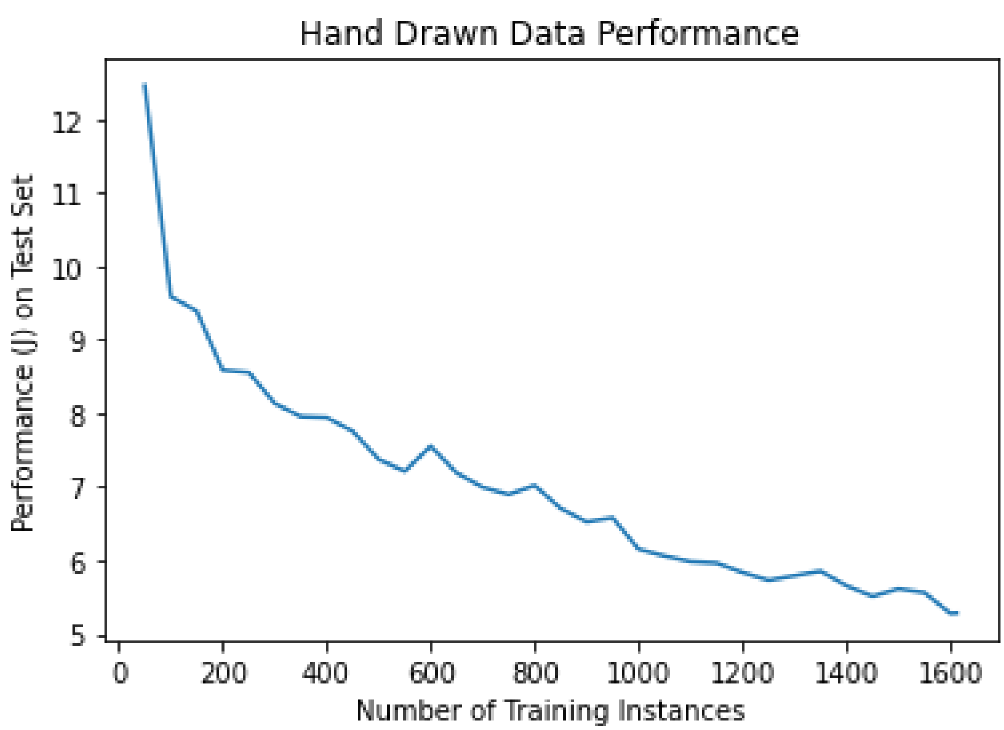
```
0  [64, 128, 10]  0.973884  0.973724
lamb = 0.1 eps = 0.0001 alpha = 5 batch_size = 50
    Architecture  Accuracy        F1
0  [64, 128, 10]  0.970489  0.970409
```

1.4 After analyzing the performance of each algorithm under different hyper-parameters, identify the best hyper-parameter setting

> From testing, using lamb = 0.1 eps = 0.0001 alpha = 5 batch_size = 50 resulted in the best performance, specifically using this architecture: [64,128,10]. For all parameters, using this architecture was best, although using 2 hidden layers with 128 neurons also yielded high performance. Lambda had the highest impact on performance, decreasing accuracy for higher values of lambda.

1.5 For each dataset, and considering the best hyper-parameter setting for each selected algorithm, construct relevant learning curves and/or graphs

[ ]:
```
dig_j,dig_count = nn.k_fold(dig_fold,dig_vals,[[64,128,10]],0.1,0.
↪0001,5,50,get_j=True)
plt.plot(dig_count,dig_j)
plt.xlabel('Number of Training Instances')
plt.ylabel('Performance (J) on Test Set')
plt.title('Hand Drawn Data Performance')
plt.show()
```



Briefly discuss and interpret these graphs

This graph shows a clear downward trend in cost as the number of training instances increase, which is to be expected of neural networks.

0.2.2 2. K-NN

2.1 discuss which algorithms you decided to test on each dataset and why > For the hand drawn numbers dataset, we decided to use K-NN due to the fact that drawings of the same numbers would likely be very close to each other in terms of euclidean distance of pixels, as well as the data all being numeric.

[ ]:
```
def test_decision_knn(train_set,test_set,vals,k_vals,fold_metrics):
    test_copy = pd.DataFrame(test_set,copy=True)
    to_guess = test_copy.drop('class',axis=1)
    predictions = pd.DataFrame(to_guess.apply(lambda row:␣
↪knn(k_vals,train_set,row), axis=1),columns=['predicted'])
    predictions['actual'] = test_set.loc[predictions.index,'class']

    for i in range(len(k_vals)):
        prec,rec,f1 = [0,0,0]
        for val in vals:
            is_targ = predictions[predictions.predicted.apply(lambda x: x[i] ==␣
↪val)]
            not_targ = predictions[predictions.predicted.apply(lambda x: x[i] !
↪= val)]
            tp = len(is_targ[is_targ['predicted'].str[i] == is_targ['actual']])
            fp = len(is_targ[is_targ['predicted'].str[i] != is_targ['actual']])
            fn = len(not_targ[not_targ.actual.apply(lambda x: x == val)])
            tn = len(not_targ[not_targ.actual.apply(lambda x: x != val)])
            this_prec = (tp/(tp+fp)) if (tp+fp) > 0 else 0
            this_rec = (tp/(tp+fn)) if (tp+fn) > 0 else 0
            f1 += (this_prec*this_rec*2)/(this_rec+this_prec) if␣
↪(this_rec+this_prec) > 0 else 0
            prec += this_prec
            rec += this_rec

        avg_f1 = f1/len(vals)
        accuracy = len(predictions[predictions['predicted'].str[i] ==␣
↪predictions['actual']])/len(test_set)
        fold_metrics[k_vals[i]].append((accuracy,avg_f1))

def knn(k_vals,data,instance):
        out = []
        distances = data.apply(lambda row: math.dist(row.
↪drop('class'),instance), axis=1)
        sorted_dist = distances.sort_values()
        for k in k_vals:
            #get k closest instances (including the input instance)
            k_neighbors = sorted_dist[:k]
```

```
            #get class value with largest number of occurences
            out.append(data.loc[k_neighbors.index,['class']]['class'].mode()[0])
        return out


np.random.seed(1)
k = 10
#function to do cross fold validation
def k_fold_jnn(fold,vals,j_vals):
    fold_metrics = defaultdict(list)
    #iterate through folds, taking turns being test fold
    for i in range(k):
        test_fold = fold[i]
        train_fold = fold[0:i]
        train_fold.extend(fold[i+1:len(fold)])
        train_data = pd.concat(train_fold)
        test_decision_knn(train_data,test_fold,vals,j_vals,fold_metrics)

    return fold_metrics

def dig_test_knn(dig_fold,dig_vals,j_vals):
    knn_res = k_fold_jnn(dig_fold,dig_vals,j_vals)
    j_dict = defaultdict(list)

    for j,perf in knn_res.items():
        avg_acc,avg_f1 = [0,0]
        for res in perf:
            avg_acc += res[0]
            avg_f1 += res[1]
        j_dict['Num Neighbors'].append(j)
        j_dict['Accuracy'].append(avg_acc/10)
        j_dict['F1'].append(avg_f1/10)

    j_table = pd.DataFrame(j_dict)
    print(j_table)
    return knn_res
```

[ ]:
```
digits = datasets.load_digits(as_frame=True)
dig_df = digits['data']
dig_df['class'] = digits['target']
dig_df = (dig_df - dig_df.min()) / (dig_df.max() - dig_df.min())
dig_df.fillna(0,inplace=True)
dig_class_0 = dig_df.loc[dig_df['class'] == 0].sample(frac=1)
dg0_split =  np.array_split(dig_class_0,k)
dig_class_1 = dig_df.loc[dig_df['class'] == 1/9].sample(frac=1)
dg1_split =  np.array_split(dig_class_1,k)
dig_class_2 = dig_df.loc[dig_df['class'] == 2/9].sample(frac=1)
```

```
dg2_split =  np.array_split(dig_class_2,k)
dig_class_3 = dig_df.loc[dig_df['class'] == 3/9].sample(frac=1)
dg3_split =  np.array_split(dig_class_3,k)
dig_class_4 = dig_df.loc[dig_df['class'] == 4/9].sample(frac=1)
dg4_split =  np.array_split(dig_class_4,k)
dig_class_5 = dig_df.loc[dig_df['class'] == 5/9].sample(frac=1)
dg5_split =  np.array_split(dig_class_5,k)
dig_class_6 = dig_df.loc[dig_df['class'] == 6/9].sample(frac=1)
dg6_split =  np.array_split(dig_class_6,k)
dig_class_7 = dig_df.loc[dig_df['class'] == 7/9].sample(frac=1)
dg7_split =  np.array_split(dig_class_7,k)
dig_class_8 = dig_df.loc[dig_df['class'] == 8/9].sample(frac=1)
dg8_split =  np.array_split(dig_class_8,k)
dig_class_9 = dig_df.loc[dig_df['class'] == 9/9].sample(frac=1)
dg9_split =  np.array_split(dig_class_9,k)

dig_vals = [0,1/9,2/9,3/9,4/9,5/9,6/9,7/9,8/9,9/9]
#list to hold folds
dig_fold = []
for i in range(k):
    this_fold =␣
 ↪[dg0_split[i],dg1_split[i],dg2_split[i],dg3_split[i],dg4_split[i],dg5_split[i],dg6_split[i]
    dig_fold.append(pd.concat(this_fold))
```

2.3 To obtain the best performance possible, you should carefully adjust the hyper-parameters of each algorithm when deployed on a dataset

[ ]:
```
j_vals = [1,10,20,30,40,50,60,70,80,90,100]
j_res = dig_test_knn(dig_fold,dig_vals,j_vals)
```

	Num Neighbors	Accuracy	F1
0	1	0.986578	0.986543
1	10	0.983294	0.983319
2	20	0.977175	0.976962
3	30	0.966028	0.965733
4	40	0.961546	0.961062
5	50	0.952601	0.952120
6	60	0.944879	0.944565
7	70	0.939832	0.939233
8	80	0.932074	0.931231
9	90	0.931525	0.930757
10	100	0.926557	0.925694

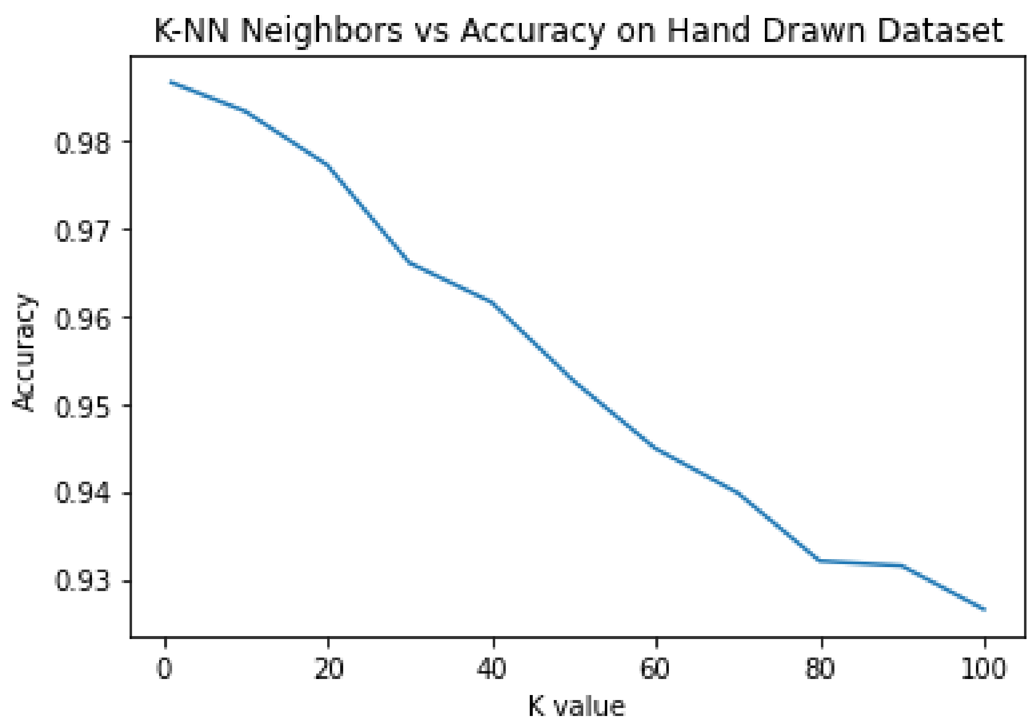
2.4 After analyzing the performance of each algorithm under different hyper-parameters, identify the best hyper-parameter setting

> From testing, using low number of neighbors yielded the best results, showing a clear decrease in accuracy as the value of K increased. The best hyper-parameter setting to use we identified was K = 1

2.5 For each dataset, and considering the best hyper-parameter setting for each selected algorithm, construct relevant learning curves and/or graphs

[ ]:
```
j_vals = []
j_acc = []
for j,perf in j_res.items():
    avg_acc,avg_f1 = [0,0]
    for res in perf:
        avg_acc += res[0]
        avg_f1 += res[1]
    j_acc.append(avg_acc/10)
    j_vals.append(j)

plt.plot(j_vals,j_acc)
plt.xlabel("K value")
plt.ylabel("Accuracy")
plt.title("K-NN Neighbors vs Accuracy on Hand Drawn Dataset")
plt.show()
```



Briefly discuss and interpret these graphs

This graph shows a clear downward accuracy as K increases. K-NN peformed very well on this dataset, which is likely due to the nature of how hand drawn numbers are stored; since each instance represents pixel values in the drawn image, we would expect that the training data point closest to our test instance in terms of euclidean distance is in fact the expected output.

0.3 Titanic Dataset

Models Used: - Neural Network - Decision tree

0.3.1 1. Neural Network

1.1 discuss which algorithms you decided to test on each dataset and why > For the titanic dataset, we decided to use a neural network due to the large sample size and mostly numeric data.

[ ]:
```
tit_df = pd.read_csv('titanic.csv')
tit_df = pd.get_dummies(tit_df.
 ↪drop('Name',axis=1),columns=['Sex'],drop_first=True,dtype=float)
fixed_df = tit_df.drop('Survived',axis=1)
fixed_df['class'] = tit_df['Survived']
fixed_df = (fixed_df - fixed_df.min()) / (fixed_df.max() - fixed_df.min())
fixed_df.fillna(0,inplace=True)
fixed_df.insert(0,'bias',1)
tit_class_0 = fixed_df.loc[fixed_df['class'] == 0].sample(frac=1)
tit_class_0['class'] = [[1,0]] * len(tit_class_0)
t0_split =  np.array_split(tit_class_0,k)
tit_class_1 = fixed_df.loc[fixed_df['class'] == 1].sample(frac=1)
tit_class_1['class'] = [[0,1]] * len(tit_class_1)
t1_split =  np.array_split(tit_class_1,k)

k = 10
tit_vals = [[1,0],[0,1]]
#list to hold folds
tit_fold = []
for i in range(k):
    this_fold = [t0_split[i],t1_split[i]]
    tit_fold.append(pd.concat(this_fold))
```

1.3 To obtain the best performance possible, you should carefully adjust the hyper-parameters of each algorithm when deployed on a dataset

[ ]:
```
tit_nn_arc = [[6,4,2],[6,8,2],[6,16,2],[6,8,16,2],[6,8,16,16,2]]
hyper_params = [[0.05,0.001,3,25],[0.05,0.001,7,25],[0.05,0.0001,3,25],[0.05,0.
 ↪0001,7,25]]
for params in hyper_params:
    ␣
 ↪dig_test(tit_fold,tit_vals,tit_nn_arc,params[0],params[1],params[2],params[3])
```

```
lamb = 0.05 eps = 0.001 alpha = 3 batch_size = 25
        Architecture  Accuracy        F1
0          [6, 4, 2]  0.784920  0.768958
1          [6, 8, 2]  0.789453  0.774124
2         [6, 16, 2]  0.792823  0.777292
3      [6, 8, 16, 2]  0.784933  0.769552
4  [6, 8, 16, 16, 2]  0.770185  0.732146
lamb = 0.05 eps = 0.001 alpha = 7 batch_size = 25
```

```
        Architecture  Accuracy        F1
0          [6, 4, 2]  0.790551  0.773858
1          [6, 8, 2]  0.783784  0.767896
2         [6, 16, 2]  0.789453  0.775258
3      [6, 8, 16, 2]  0.788291  0.772513
4  [6, 8, 16, 16, 2]  0.791674  0.775615
lamb = 0.05 eps = 0.0001 alpha = 3 batch_size = 25
        Architecture  Accuracy        F1
0          [6, 4, 2]  0.784908  0.768253
1          [6, 8, 2]  0.793960  0.778619
2         [6, 16, 2]  0.787193  0.772083
3      [6, 8, 16, 2]  0.792798  0.776038
4  [6, 8, 16, 16, 2]  0.791687  0.775201
lamb = 0.05 eps = 0.0001 alpha = 7 batch_size = 25
        Architecture  Accuracy        F1
0          [6, 4, 2]  0.791713  0.775348
1          [6, 8, 2]  0.793973  0.779222
2         [6, 16, 2]  0.779263  0.763847
3      [6, 8, 16, 2]  0.789440  0.774200
4  [6, 8, 16, 16, 2]  0.783771  0.768099
```

[ ]:
```python
tit_nn_arc =␣
 ↪[[6,8,2],[6,128,2],[6,16,2],[6,128,128,2],[6,16,32,16,2],[6,16,32,32,16,2]]
hyper_params = [[0.1,0.0001,2,20],[0.1,0.0001,7,20],[0.1,0.0001,2,25],[0.1,0.
 ↪0001,7,25]]
for params in hyper_params:
    ␣
 ↪dig_test(tit_fold,tit_vals,tit_nn_arc,params[0],params[1],params[2],params[3])
```

```
lamb = 0.1 eps = 0.0001 alpha = 2 batch_size = 20
              Architecture  Accuracy        F1
0                [6, 8, 2]  0.787155  0.770646
1              [6, 128, 2]  0.753293  0.747549
2               [6, 16, 2]  0.787193  0.771480
3         [6, 128, 128, 2]  0.755528  0.751763
4       [6, 16, 32, 16, 2]  0.788291  0.773472
5   [6, 16, 32, 32, 16, 2]  0.783809  0.769452
lamb = 0.1 eps = 0.0001 alpha = 7 batch_size = 20
              Architecture  Accuracy        F1
0                [6, 8, 2]  0.788329  0.773561
1              [6, 128, 2]  0.685758  0.684840
2               [6, 16, 2]  0.777041  0.763296
3         [6, 128, 128, 2]  0.666302  0.660193
4       [6, 16, 32, 16, 2]  0.767976  0.756175
5   [6, 16, 32, 32, 16, 2]  0.760073  0.724336
lamb = 0.1 eps = 0.0001 alpha = 2 batch_size = 25
              Architecture  Accuracy        F1
```

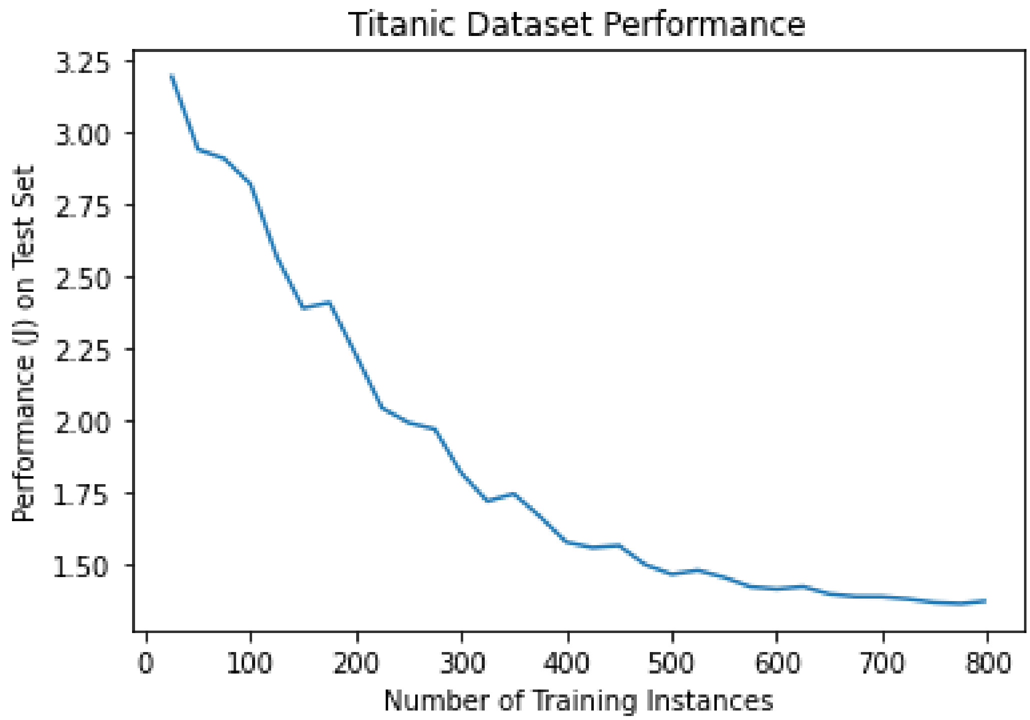
```
0                [6, 8, 2]  0.786031  0.769922
1              [6, 128, 2]  0.782533  0.770359
2               [6, 16, 2]  0.789453  0.773901
3          [6, 128, 128, 2]  0.783721  0.776729
4       [6, 16, 32, 16, 2]  0.787193  0.770604
5   [6, 16, 32, 32, 16, 2]  0.788316  0.773110
lamb = 0.1 eps = 0.0001 alpha = 7 batch_size = 25
             Architecture  Accuracy        F1
0                [6, 8, 2]  0.790602  0.775523
1              [6, 128, 2]  0.757826  0.752779
2               [6, 16, 2]  0.780375  0.765427
3          [6, 128, 128, 2]  0.746664  0.742149
4       [6, 16, 32, 16, 2]  0.786082  0.771820
5   [6, 16, 32, 32, 16, 2]  0.783784  0.767972
```

1.4 After analyzing the performance of each algorithm under different hyper-parameters, identify the best hyper-parameter setting

From testing, using the best accuracy we could achieve was with lamb = 0.05 eps = 0.0001 alpha = 3 batch_size = 25 using an architecture with one hidden layer containing 8 neurons: [6,8,2]

1.5 For each dataset, and considering the best hyper-parameter setting for each selected algorithm, construct relevant learning curves and/or graphs

[ ]:
```
tit_j,tit_count = nn.k_fold(tit_fold,tit_vals,[[6,8,2]],0.05,0.
 ↪0001,3,25,get_j=True)
plt.plot(tit_count,tit_j)
plt.xlabel('Number of Training Instances')
plt.ylabel('Performance (J) on Test Set')
plt.title('Titanic Dataset Performance')
plt.show()
```



Briefly discuss and interpret these graphs

> This graph shows an reduction in cost as training instances increases, which is expected of neural nets. The general performance of this model on the titanic data set is quite low ~80% showing similar performance to other models. This could be due to the fact that determining survival is not easy to determing based on the information given.

0.3.2 2. Decision Tree

2.1 discuss which algorithms you decided to test on each dataset and why > For the titanic dataset, we decided to use a decision due to the large sample size. For a smaller dataset, we would have likely used a random forest instead to avoid overfitting.

[ ]:
```
titanic_data_read = pd.read_csv('titanic.csv')
titanic_data = pd.get_dummies(titanic_data_read.
 ↪drop('Name',axis=1),columns=['Sex'],drop_first=True)
titanic_data = titanic_data.drop('Survived',axis=1)
titanic_data['class'] = titanic_data_read['Survived']
titanic_attr = defaultdict(list)
attr_list = list(titanic_data.columns.values)
attr_list.remove('class')
titanic_targets = [0,1]

k = 10
```

```
#split data by class into k groups then combine into folds
tit_class_0 = titanic_data.loc[titanic_data['class'] == 0].sample(frac=1)
td0_split = np.array_split(tit_class_0,k)
tit_class_1 = titanic_data.loc[titanic_data['class'] == 1].sample(frac=1)
td1_split = np.array_split(tit_class_1,k)

titanic_fold = []
for i in range(k):
    this_fold = [td0_split[i],td1_split[i]]
    titanic_fold.append(pd.concat(this_fold))
```

[ ]:
```
def␣
↪decision_tree_knn(titanic_fold,attr_list,titanic_attr,titanic_targets,depth,min_size_split,
↪
    #for depth in max_depth_arr:
    fold_metrics_titanic = rf.
↪k_fold(titanic_fold,attr_list,titanic_attr,titanic_targets,[1],do_forest =␣
↪False, max_depth=depth,min_size_split=min_size_split,maj_prop=maj_prop)
    n_acc = []
    n_prec = []
    n_rec = []
    n_f1 = []

    for n,perf in fold_metrics_titanic.items():
        avg_accuracy,avg_prec,avg_rec,avg_f1 = [0,0,0,0]
        for res in perf:
            avg_accuracy += res[0]
            avg_prec += res[1]
            avg_rec += res[2]
            avg_f1 += res[3]
        n_acc.append(avg_accuracy/10)
        n_prec.append(avg_prec/10)
        n_rec.append(avg_rec/10)
        n_f1.append(avg_f1/10)

    if for_graph:
        return n_acc

    print(f'max_depth: {depth} min_size_split: {min_size_split} maj prop:␣
↪{maj_prop}')
    print("Accuracy: ", n_acc)
    print("Precision", n_prec)
    print("Recall", n_rec)
    print("F1", n_f1)
```

2.3 To obtain the best performance possible, you should carefully adjust the hyper-parameters of each algorithm when deployed on a dataset

```
[ ]: hyper_params = [[6,10,0.875],[7,10,0.875],[8,10,0.875],[9,10,0.875],[10,10,0.
     ↪875]]
     for params in hyper_params:
        ␣
     ↪decision_tree_knn(titanic_fold,attr_list,titanic_attr,titanic_targets,params[0],params[1],p
```

```
max_depth: 6 min_size_split: 10 maj prop: 0.875
Accuracy:  [0.7833546135512428]
Precision [0.7763838658091624]
Recall [0.7592722745663922]
F1 [0.7643430828668846]
max_depth: 7 min_size_split: 10 maj prop: 0.875
Accuracy:  [0.8094784927930995]
Precision [0.8096850489334951]
Recall [0.7895017400899753]
F1 [0.7928574015108906]
max_depth: 8 min_size_split: 10 maj prop: 0.875
Accuracy:  [0.7867889569855862]
Precision [0.7831971411783987]
Recall [0.7622012845542256]
F1 [0.7676562726774813]
max_depth: 9 min_size_split: 10 maj prop: 0.875
Accuracy:  [0.7936196231982748]
Precision [0.7936733247493835]
Recall [0.7711173358232181]
F1 [0.7749397219374048]
max_depth: 10 min_size_split: 10 maj prop: 0.875
Accuracy:  [0.8059556236522528]
Precision [0.8055203103948367]
Recall [0.7842510539569363]
F1 [0.7893199599303956]
```

```
[ ]: hyper_params = [[7,8,0.875],[7,10,0.875],[7,12,0.875],[7,14,0.875],[7,16,0.875]]
     for params in hyper_params:
        ␣
     ↪decision_tree_knn(titanic_fold,attr_list,titanic_attr,titanic_targets,params[0],params[1],p
```

```
max_depth: 7 min_size_split: 8 maj prop: 0.875
Accuracy:  [0.7868525139030756]
Precision [0.7815060899650035]
Recall [0.7611036131624366]
F1 [0.7666713072748895]
max_depth: 7 min_size_split: 10 maj prop: 0.875
Accuracy:  [0.8013715809783226]
Precision [0.8034633941103706]
Recall [0.7734497637438814]
F1 [0.7809410193867494]
```

```
max_depth: 7 min_size_split: 12 maj prop: 0.875
Accuracy:  [0.8059805924412666]
Precision [0.8093006763334332]
Recall [0.7719548425430782]
F1 [0.7838276282816296]
max_depth: 7 min_size_split: 14 maj prop: 0.875
Accuracy:  [0.7902729542617184]
Precision [0.785550279759695]
Recall [0.7693785190844015]
F1 [0.7737302132279205]
max_depth: 7 min_size_split: 16 maj prop: 0.875
Accuracy:  [0.7845928384973329]
Precision [0.7778822794683296]
Recall [0.7597964236199531]
F1 [0.76500591853942]
```

[ ]:
```
hyper_params = [[7,8,0.85],[7,8,0.875],[7,8,0.9],[7,8,0.925],[7,8,0.95],]
for params in hyper_params:
    decision_tree_knn(titanic_fold,attr_list,titanic_attr,titanic_targets,params[0],params[1],p
```

```
max_depth: 7 min_size_split: 8 maj prop: 0.85
Accuracy:  [0.7902494041538984]
Precision [0.7848700728138863]
Recall [0.7660349998585293]
F1 [0.7717361595535912]
max_depth: 7 min_size_split: 8 maj prop: 0.875
Accuracy:  [0.8094529565316083]
Precision [0.810839575056327]
Recall [0.783710352828]
F1 [0.7900802855947241]
max_depth: 7 min_size_split: 8 maj prop: 0.9
Accuracy:  [0.8049826920894336]
Precision [0.8052916894827126]
Recall [0.7819602184308068]
F1 [0.7873162888353912]
max_depth: 7 min_size_split: 8 maj prop: 0.925
Accuracy:  [0.7844535240040859]
Precision [0.7782514870143593]
Recall [0.760709334238746]
F1 [0.7652106824381794]
max_depth: 7 min_size_split: 8 maj prop: 0.95
Accuracy:  [0.7981903302689819]
Precision [0.7996295711975768]
Recall [0.768829612653142]
F1 [0.7773198110970176]
```

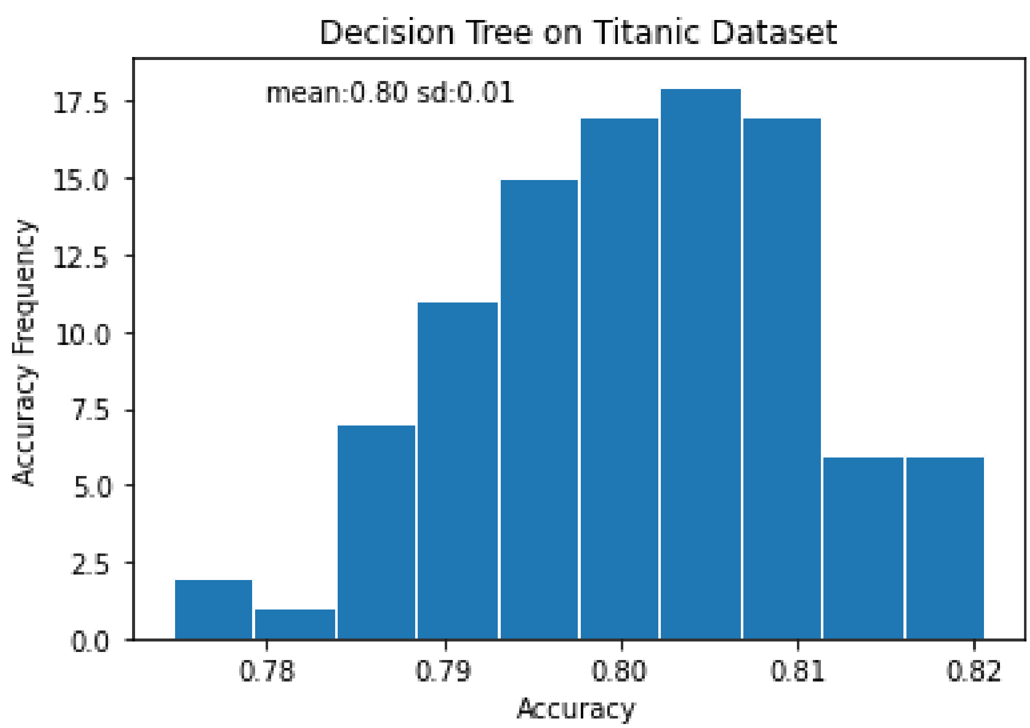
2.4 After analyzing the performance of each algorithm under different hyper-parameters, identify the best hyper-parameter setting

From testing, using the best accuracy we could achieve was with max depth of 7, minimum size split of 8, and majority prop value of 0.9

2.5 For each dataset, and considering the best hyper-parameter setting for each selected algorithm, construct relevant learning curves and/or graphs

```
[ ]: accuracy_arr = []
for i in range(100):
    accuracy_arr.
 ↪append(decision_tree_knn(titanic_fold,attr_list,titanic_attr,titanic_targets,7,8,0.
 ↪9,True))
```

```
[ ]: num_bins = 10
fixed_arr = [acc[0] for acc in accuracy_arr]
plt.hist(fixed_arr,bins=num_bins,edgecolor='white',linewidth=1)
plt.ylabel("Accuracy Frequency")
plt.xlabel("Accuracy")
plt.title("Decision Tree on Titanic Dataset")
plt.text(0.78,17.5,f'mean:{np.mean(accuracy_arr):.2f} sd:{np.std(accuracy_arr):.
 ↪2f}')
plt.show()
```



Briefly discuss and interpret these graphs

This graph shows the average accuracy over 100 iterations of the decision tree, yielding a mean of 80% accuracy with a standard deviation of about 1%. This performance is similar to the performance of neural nets, leading us to believe that the low accuracy is due to the nature of predicting titanic survival being hard (being lucky probably played a role in survival).

0.4 Loan Eligibility Prediction Dataset

Models Used: - K-NN - Random Forests

0.4.1 1. K-NN

1.1 discuss which algorithms you decided to test on each dataset and why > For the loan dataset, we decided to use K-NN because we believed that people who are most similar to eachother based on the attributes would recieve the same loan status.

[ ]:
```
loan_df = pd.read_csv('loan.csv')
dum_df = pd.get_dummies(loan_df.
 ↪drop('Loan_ID',axis=1),columns=['Gender','Married','Education','Self_Employed','Property_Ar
fixed_df = dum_df.drop('Loan_Status_Y',axis=1)
fixed_df.loc[fixed_df['Dependents'] == '3+','Dependents'] = 3
fixed_df['Dependents'] = pd.to_numeric(fixed_df['Dependents'])
fixed_df['class'] = dum_df['Loan_Status_Y']
fixed_df = (fixed_df - fixed_df.min()) / (fixed_df.max() - fixed_df.min())
fixed_df.fillna(0,inplace=True)
loan_class_0 = fixed_df.loc[fixed_df['class'] == 0].sample(frac=1)
l0_split =  np.array_split(loan_class_0,k)
loan_class_1 = fixed_df.loc[fixed_df['class'] == 1].sample(frac=1)
l1_split =  np.array_split(loan_class_1,k)


k = 10
loan_vals = [0,1]
#list to hold folds
loan_fold = []
for i in range(k):
    this_fold = [l0_split[i],l1_split[i]]
    loan_fold.append(pd.concat(this_fold))
```

[ ]:
```
j_vals = [1,5,10,15,20,30,40,50,60,70]
j_res = dig_test_knn(loan_fold,loan_vals,j_vals)
```

	Num Neighbors	Accuracy	F1
0	1	0.699818	0.630189
1	5	0.773054	0.682017
2	10	0.770973	0.676292
3	15	0.762636	0.628743
4	20	0.764763	0.636746
5	30	0.762458	0.620200

6	40	0.739671	0.563968
7	50	0.747831	0.575239
8	60	0.731245	0.530473
9	70	0.718657	0.495590

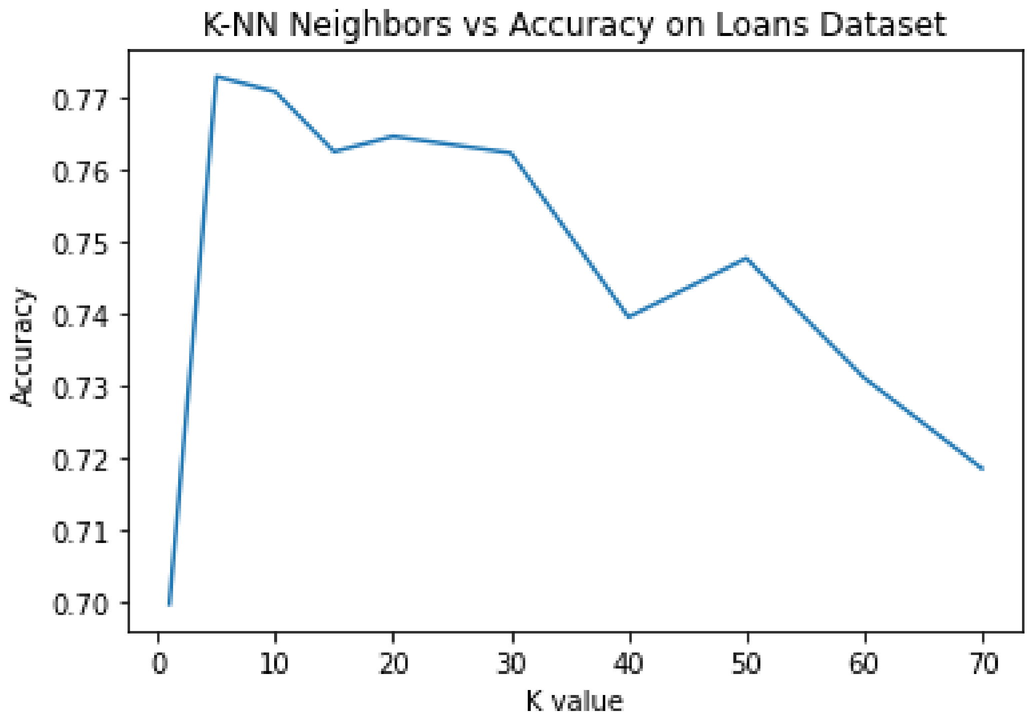
1.4 After analyzing the performance of each algorithm under different hyper-parameters, identify the best hyper-parameter setting

From testing, using low number of neighbors yielded the best results, showing a decreasing trend as K grows larger than 5. The best hyper-parameter setting to use we identified was K = 5

1.5 For each dataset, and considering the best hyper-parameter setting for each selected algorithm, construct relevant learning curves and/or graphs

[ ]:
```
j_vals = []
j_acc = []
for j,perf in j_res.items():
    avg_acc,avg_f1 = [0,0]
    for res in perf:
        avg_acc += res[0]
        avg_f1 += res[1]
    j_acc.append(avg_acc/10)
    j_vals.append(j)

plt.plot(j_vals,j_acc)
plt.xlabel("K value")
plt.ylabel("Accuracy")
plt.title("K-NN Neighbors vs Accuracy on Loans Dataset")
plt.show()
```



Briefly discuss and interpret these graphs

This graph shows a large jump in accuracy from K = 1 to K = 5. For all K > 5, there is a clear downward trend in accuracy as K increases. This model did not perform well for the dataset, likely due to the fact that the loans dataset contains a large number of categorical variables.

0.4.2 2. Random Forest

2.1 discuss which algorithms you decided to test on each dataset and why > For the loan dataset, we decided to use Random Forest because the dataset is not too large and contains a lot of categorical variables that a Random Forest would be more equipped to handle as opposed neural net.

[ ]:
```
k = 10
loan_df = pd.read_csv('loan.csv')
dum_df = pd.get_dummies(loan_df.
 ↪drop('Loan_ID',axis=1),columns=['Gender','Married','Education','Self_Employed','Property_Ar
fixed_df = dum_df.drop('Loan_Status_Y',axis=1)
fixed_df['class'] = dum_df['Loan_Status_Y']
fixed_df.loc[fixed_df['Dependents'] == '3+','Dependents'] = 3
fixed_df['Dependents'] = pd.to_numeric(fixed_df['Dependents'])
fixed_df.fillna(0,inplace=True)
loan_class_0 = fixed_df.loc[fixed_df['class'] == 0].sample(frac=1)
l0_split =  np.array_split(loan_class_0,k)
loan_class_1 = fixed_df.loc[fixed_df['class'] == 1].sample(frac=1)
```

```
l1_split =  np.array_split(loan_class_1,k)
attr_list = list(fixed_df.columns.values)
attr_list.remove('class')
loan_attr = defaultdict(list)

loan_vals = [0,1]
#list to hold folds
loan_fold = []
for i in range(k):
    this_fold = [l0_split[i],l1_split[i]]
    loan_fold.append(pd.concat(this_fold))
```

[ ]:
```
def␣
↪random_forest_kfolds(titanic_fold,attr_list,titanic_attr,titanic_targets,depth,min_size_spl
↪
    #for depth in max_depth_arr:
    fold_metrics_titanic = rf.
↪k_fold(titanic_fold,attr_list,titanic_attr,titanic_targets,[1,10,20,30,40],do_forest␣
↪= True, max_depth=depth,min_size_split=min_size_split,maj_prop=maj_prop)
    n_acc = []
    n_prec = []
    n_rec = []
    n_f1 = []

    for n,perf in fold_metrics_titanic.items():
        avg_accuracy,avg_prec,avg_rec,avg_f1 = [0,0,0,0]
        for res in perf:
            avg_accuracy += res[0]
            avg_prec += res[1]
            avg_rec += res[2]
            avg_f1 += res[3]
        n_acc.append(avg_accuracy/10)
        n_prec.append(avg_prec/10)
        n_rec.append(avg_rec/10)
        n_f1.append(avg_f1/10)

    if for_graph:
        return [n_acc, n_prec, n_rec, n_f1]

    print(f'max_depth: {depth} min_size_split: {min_size_split} maj prop:␣
↪{maj_prop}')
    print("Accuracy: ", n_acc)
    print("Precision", n_prec)
    print("Recall", n_rec)
    print("F1", n_f1)
```

2.3 To obtain the best performance possible, you should carefully adjust the hyper-parameters of each algorithm when deployed on a dataset

[ ]:
```
hyper_params = [[7,25,0.875],[8,25,0.875],[9,25,0.875]]
for params in hyper_params:
    random_forest_kfolds(loan_fold,attr_list,loan_attr,loan_vals,params[0],params[1],params[2])
```

```
max_depth: 7 min_size_split: 25 maj prop: 0.875
Accuracy:  [0.7332455854682298, 0.8042652699377625, 0.8063504125054278,
0.8084780720798959, 0.8063947387465624]
Precision [0.7046930426910638, 0.8347106631804142, 0.8445156085816443,
0.8519023681635259, 0.8462708350624807]
Recall [0.6489253883371531, 0.7016042780748662, 0.7012566844919785,
0.7027718360071301, 0.7012566844919786]
F1 [0.6527785071785073, 0.7235433055824648, 0.7234965645229028,
0.7256841141077126, 0.723720853538139]
max_depth: 8 min_size_split: 25 maj prop: 0.875
Accuracy:  [0.7523040599218411, 0.8001004125054276, 0.8022262628455639,
0.8064372557533653, 0.8084780720798959]
Precision [0.7158296334011872, 0.8181794045326601, 0.835407836806992,
0.8467973546133905, 0.8519023681635259]
Recall [0.6570473644003055, 0.7005856888209829, 0.6982709447415331,
0.701301247771836, 0.7027718360071301]
F1 [0.6627477229128639, 0.7196524939981486, 0.7193520423732469,
0.7236592414089269, 0.7256841141077126]
max_depth: 9 min_size_split: 25 maj prop: 0.875
Accuracy:  [0.7255590534085974, 0.8063947387465624, 0.8084780720798959,
0.8084780720798959, 0.8063947387465624]
Precision [0.7017967301904651, 0.8480170713834922, 0.8470360748960697,
0.8519023681635259, 0.8462708350624807]
Recall [0.6504787369493252, 0.7012566844919785, 0.7045900178253119,
0.7027718360071301, 0.7012566844919786]
F1 [0.6548778164753988, 0.7236485159031005, 0.727624179677633,
0.7256841141077126, 0.723720853538139]
```

[ ]:
```
hyper_params = [[7,15,0.875],[7,20,0.875],[7,30,0.875]]
for params in hyper_params:
    random_forest_kfolds(loan_fold,attr_list,loan_attr,loan_vals,params[0],params[1],params[2])
```

```
max_depth: 7 min_size_split: 15 maj prop: 0.875
Accuracy:  [0.7686712983065568, 0.7918503039513679, 0.8043095961788971,
0.8042670791720944, 0.8042670791720944]
Precision [0.7483736008976702, 0.8046120470231584, 0.837787788013965,
0.8406303118016106, 0.8405128503571275]
Recall [0.6890947288006112, 0.6944652406417111, 0.7016042780748661,
0.699741532976827, 0.6997415329768272]
F1 [0.7006680238107059, 0.7131978418130868, 0.7226213141960297,
0.7214609663182906, 0.7213769150720326]
```

```
max_depth: 7 min_size_split: 20 maj prop: 0.875
Accuracy:  [0.7399298017079172, 0.8063486032710957, 0.793804277029961,
0.8063947387465624, 0.8042670791720944]
Precision [0.6986365784856261, 0.8327851021999753, 0.8052895261762474,
0.8432921799021749, 0.8406303118016106]
Recall [0.6504443595620065, 0.7069938884644766, 0.6940285204991088,
0.7012566844919785, 0.699741532976827]
F1 [0.6530527934420565, 0.7287410747802341, 0.7132503868828037,
0.7237392519523242, 0.7214609663182906]
max_depth: 7 min_size_split: 30 maj prop: 0.875
Accuracy:  [0.7605523592415689, 0.8001411202778984, 0.8105614054132293,
0.8064372557533653, 0.8084780720798959]
Precision [0.7374214018329306, 0.8405882655994233, 0.8528159894259844,
0.8467973546133905, 0.8519023681635259]
Recall [0.6696218487394958, 0.6926878023936848, 0.7061051693404634,
0.701301247771836, 0.7027718360071301]
F1 [0.6803950393337341, 0.7141127098212994, 0.7298344900475623,
0.7236592414089269, 0.7256841141077126]
```

[ ]:
```python
hyper_params = [[7,25,0.875],[7,25,0.9],[7,25,0.925]]
for params in hyper_params:
    ␣
 ↪random_forest_kfolds(loan_fold,attr_list,loan_attr,loan_vals,params[0],params[1],params[2])
```

```
max_depth: 7 min_size_split: 25 maj prop: 0.875
Accuracy:  [0.7625931755680997, 0.8041784266898249, 0.7959301273700969,
0.8064372557533653, 0.8084780720798959]
Precision [0.7392302373578243, 0.8340576920251964, 0.8227431330896566,
0.8467973546133905, 0.8519023681635259]
Recall [0.6766959511077159, 0.7015597147950089, 0.6937700534759359,
0.701301247771836, 0.7027718360071301]
F1 [0.6883314284671587, 0.7232242398151444, 0.7130331155266963,
0.7236592414089269, 0.7256841141077126]
max_depth: 7 min_size_split: 25 maj prop: 0.9
Accuracy:  [0.7439300188160372, 0.7978841004486902, 0.8063504125054278,
0.8063504125054278, 0.8084780720798959]
Precision [0.7070880193451552, 0.8227063405223761, 0.8445156085816443,
0.8461443834581728, 0.8519023681635259]
Recall [0.6560695187165775, 0.6951960784313725, 0.7012566844919785,
0.7012566844919786, 0.7027718360071301]
F1 [0.6635845939039583, 0.7146295698935496, 0.7234965645229028,
0.7233401756416065, 0.7256841141077126]
max_depth: 7 min_size_split: 25 maj prop: 0.925
Accuracy:  [0.7669353379649733, 0.7981039224200319, 0.8000986032710957,
0.8084780720798959, 0.8084780720798959]
Precision [0.7478979776348197, 0.8267966025758143, 0.8276247321099846,
0.8519023681635259, 0.8519023681635259]
```

```
Recall [0.6945543672014259, 0.6934670231729055, 0.6967557932263814,
0.7027718360071301, 0.7027718360071301]
F1 [0.7071246084324047, 0.7139696313540707, 0.7173487113138222,
0.7256841141077126, 0.7256841141077126]
```

2.4 After analyzing the performance of each algorithm under different hyper-parameters, identify the best hyper-parameter setting

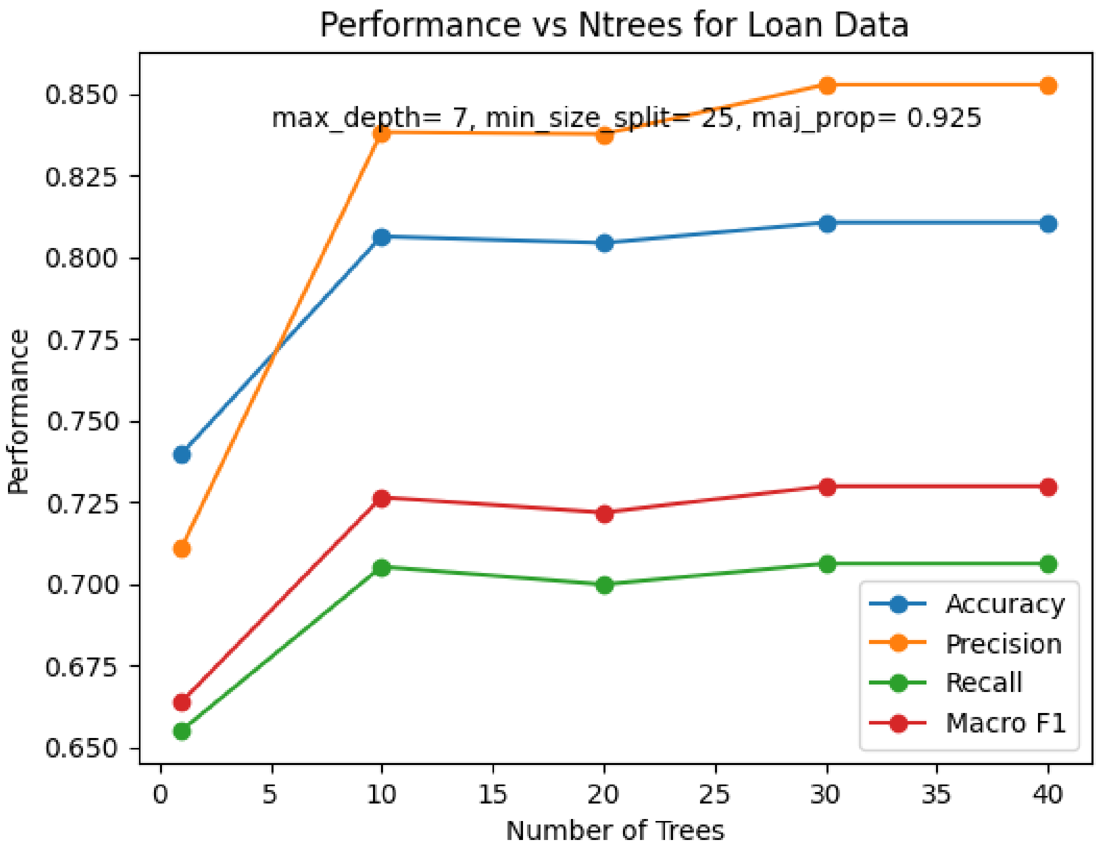
> From testing, using the best accuracy we could achieve was with max depth of 7, minimum size split of 25, and majority prop value of 0.925, ntree value of 30

2.5 For each dataset, and considering the best hyper-parameter setting for each selected algorithm, construct relevant learning curves and/or graphs

[ ]:
```
hyper_params = [[7,25,0.925]]
n_acc = []
n_prec = []
n_rec = []
n_f1 = []
for params in hyper_params:
    results =␣
 ↪random_forest_kfolds(loan_fold,attr_list,loan_attr,loan_vals,params[0],params[1],params[2],
 ↪True)
    n_acc = results[0]
    n_prec = results[1]
    n_rec = results[2]
    n_f1 =results[3]



nvals = [1,10,20,30,40]
plt.title('Performance vs Ntrees for Loan Data')
plt.xlabel('Number of Trees')
plt.ylabel('Performance')
plt.plot(nvals,n_acc,label='Accuracy',marker='o')
plt.plot(nvals,n_prec,label='Precision',marker='o')
plt.plot(nvals,n_rec,label='Recall',marker='o')
plt.plot(nvals,n_f1,label='Macro F1',marker='o')
plt.legend()
plt.text(5,0.84,'max_depth= 7, min_size_split= 25, maj_prop= 0.925')
plt.show()
```



Briefly discuss and interpret these graphs

> This graph shows the accuracy, precision, recall, and f1 values for different number of trees from 1 tree to 40 trees in each forest. As seen the accuracy and F1 plateau from 10-40 trees and there really isn't much improvement. The accuracy flattens out at around 0.8 and the f1 value flattens out at around 0.725. One reason why the results are low might be because of the amount of features that are categorical, so a random forest would have trouble with the splits inside the tree.

0.5 Oxford Parkingson's Disease Detection

Models Used: - K-NN - Random Forest

0.5.1 1. K-NN

1.1 discuss which algorithms you decided to test on each dataset and why > For the parkinsons dataset, we decided to use K-NN because all of the data is numeric.

[ ]:
```
park_df = pd.read_csv('parkinsons.csv')

park_fix = park_df.drop('Diagnosis',axis=1)
park_fix['class'] = park_df['Diagnosis']
```

```
park_fix = (park_fix - park_fix.min()) / (park_fix.max() - park_fix.min())
park_fix.fillna(0,inplace=True)
park_class_0 = park_fix.loc[park_fix['class'] == 0].sample(frac=1)
p0_split =  np.array_split(park_class_0,k)
park_class_1 = park_fix.loc[park_fix['class'] == 1].sample(frac=1)
p1_split =  np.array_split(park_class_1,k)

k = 10
park_vals = [0,1]
#list to hold folds
park_fold = []
for i in range(k):
    this_fold = [p0_split[i],p1_split[i]]
    park_fold.append(pd.concat(this_fold))
```

1.4 After analyzing the performance of each algorithm under different hyper-parameters, identify the best hyper-parameter setting

> From testing, using low number of neighbors yielded the best results, showing a clear decrease in accuracy as the value of K increased. The best hyper-parameter setting to use we identified was K = 1

1.5 For each dataset, and considering the best hyper-parameter setting for each selected algorithm, construct relevant learning curves and/or graphs

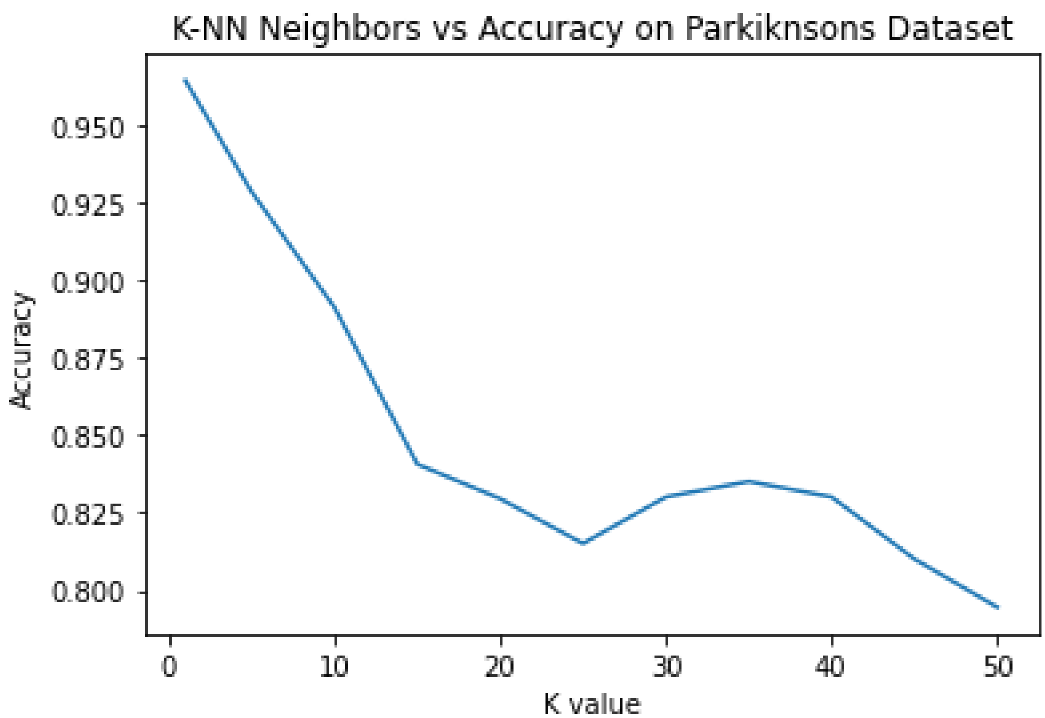
[ ]:
```
j_vals = [1,5,10,15,20,25,30,35,40,45,50]
j_res = dig_test_knn(park_fold,park_vals,j_vals)
```

	Num Neighbors	Accuracy	F1
0	1	0.964211	0.954831
1	5	0.928392	0.901292
2	10	0.891170	0.845451
3	15	0.840614	0.728745
4	20	0.829503	0.676902
5	25	0.815058	0.636132
6	30	0.830058	0.670143
7	35	0.835058	0.672254
8	40	0.830058	0.663797
9	45	0.810058	0.609038
10	50	0.794503	0.564036

[ ]:
```
j_vals = []
j_acc = []
for j,perf in j_res.items():
    avg_acc,avg_f1 = [0,0]
    for res in perf:
        avg_acc += res[0]
        avg_f1 += res[1]
    j_acc.append(avg_acc/10)
```

```
    j_vals.append(j)

plt.plot(j_vals,j_acc)
plt.xlabel("K value")
plt.ylabel("Accuracy")
plt.title("K-NN Neighbors vs Accuracy on Parkiknsons Dataset")
plt.show()
```



Briefly discuss and interpret these graphs

> This graph shows a clear downward accuracy as K increases. K-NN peformed very well on this dataset, which is likely due to the fact that the parkinsons data is all continuous and numeric.

0.5.2 2. Random Forest

2.1 discuss which algorithms you decided to test on each dataset and why > For the Parkinsons dataset, we decided to use Random Forest because the dataset is not large so a Random Forest will allow us to achieve multiple decisions on the same small dataset. Also the Parkinsons dataset has a high number of features so Random Forest will perform better with all the features.

[ ]:
```
k = 10
park_df = pd.read_csv('parkinsons.csv')

park_fix = park_df.drop('Diagnosis',axis=1)
```

```
park_fix['class'] = park_df['Diagnosis']
park_fix.fillna(0,inplace=True)
park_class_0 = park_fix.loc[park_fix['class'] == 0].sample(frac=1)
p0_split =  np.array_split(park_class_0,k)
park_class_1 = park_fix.loc[park_fix['class'] == 1].sample(frac=1)
p1_split =  np.array_split(park_class_1,k)
attr_list = list(park_fix.columns.values)
attr_list.remove('class')
park_attr = defaultdict(list)


park_vals = [0,1]
#list to hold folds
park_fold = []
for i in range(k):
    this_fold = [p0_split[i],p1_split[i]]
    park_fold.append(pd.concat(this_fold))
```

2.3 To obtain the best performance possible, you should carefully adjust the hyper-parameters of each algorithm when deployed on a dataset

[ ]:
```
hyper_params = [[7,10,0.875],[8,10,0.875],[9,10,0.875]]
for params in hyper_params:
    ␣
 ↪random_forest_kfolds(park_fold,attr_list,park_attr,park_vals,params[0],params[1],params[2])
```

```
max_depth: 7 min_size_split: 10 maj prop: 0.875
Accuracy:  [0.8295029239766082, 0.8900584795321638, 0.90140350877193,
0.8972514619883041, 0.9019883040935672]
Precision [0.7631617647058826, 0.8730164565826332, 0.8843820028011203,
0.888876050420168, 0.8826425249587014]
Recall [0.7404761904761906, 0.816547619047619, 0.8326190476190476,
0.8317857142857144, 0.8440476190476192]
F1 [0.7401040112094062, 0.8307172882888956, 0.8473610141352076,
0.8488489482955556, 0.854911470697761]
max_depth: 8 min_size_split: 10 maj prop: 0.875
Accuracy:  [0.8416959064327484, 0.8964035087719298, 0.876169590643275,
0.917514619883041, 0.9128070175438596]
Precision [0.7984215865833514, 0.877450980392157, 0.8554429271708685,
0.903423202614379, 0.9176984741264926]
Recall [0.7729761904761906, 0.8359523809523809, 0.8026190476190477,
0.8586904761904762, 0.851190476190476]
F1 [0.7703261207029282, 0.8454683638519844, 0.8188793655207727,
0.8715494191210265, 0.8656458772793506]
max_depth: 9 min_size_split: 10 maj prop: 0.875
Accuracy:  [0.8023099415204678, 0.8858771929824563, 0.9230701754385967,
0.88640350877193, 0.8972514619883041]
Precision [0.7142298265460031, 0.8696606334841629, 0.9204236694677871,
0.8615482026143791, 0.916061569364588]
```

```
Recall [0.714404761904762, 0.8223809523809523, 0.8711904761904762,
0.8092857142857144, 0.8097619047619048]
F1 [0.7078098370082521, 0.8299881883349624, 0.8855577599181602,
0.8251415938477951, 0.8352437991325562]
```

[ ]:
```
hyper_params = [[8,10,0.875],[8,5,0.875],[8,15,0.875], [8,20,0.875]]
for params in hyper_params:
    ␣
  ↪random_forest_kfolds(park_fold,attr_list,park_attr,park_vals,params[0],params[1],params[2])
```

```
max_depth: 8 min_size_split: 10 maj prop: 0.875
Accuracy:  [0.7886549707602339, 0.9011403508771931, 0.9219883040935672,
0.9230701754385965, 0.9069883040935673]
Precision [0.7296319040436687, 0.8820635534915722, 0.929229826546003,
0.9227941176470589, 0.9056881598793364]
Recall [0.7133333333333334, 0.8083333333333333, 0.8617857142857142,
0.8689285714285715, 0.8451190476190475]
F1 [0.7020600537455376, 0.8191361370196105, 0.8822356981944022,
0.8839208551562556, 0.8627374903090976]
max_depth: 8 min_size_split: 5 maj prop: 0.875
Accuracy:  [0.8614327485380118, 0.8928070175438597, 0.9228070175438597,
0.9078070175438595, 0.9016959064327486]
Precision [0.8149124649859945, 0.8918949026979213, 0.9356939223057645,
0.8917670401493931, 0.8925560224089637]
Recall [0.8146428571428572, 0.8245238095238095, 0.8645238095238096,
0.8542857142857143, 0.832857142857143]
F1 [0.8057814369538507, 0.8355066426212062, 0.8776935913633551,
0.8616753792184826, 0.8519490019775059]
max_depth: 8 min_size_split: 15 maj prop: 0.875
Accuracy:  [0.8447953216374268, 0.9022514619883042, 0.9016959064327486,
0.907514619883041, 0.9125146198830411]
Precision [0.805625, 0.9042279411764707, 0.8907074175824174, 0.9003869047619049,
0.9023967086834732]
Recall [0.7858333333333334, 0.8266666666666668, 0.8417857142857142,
0.8520238095238095, 0.8553571428571429]
F1 [0.7841144558289213, 0.847622105369602б, 0.8581848438300052,
0.8651798668928814, 0.8699371790097595]
max_depth: 8 min_size_split: 20 maj prop: 0.875
Accuracy:  [0.7995029239766082, 0.8597660818713451, 0.8919883040935673,
0.87140350877193, 0.8753216374269007]
Precision [0.7149855455002514, 0.8109751400560224, 0.8817298265460028,
0.8489449112978527, 0.8359728672170623]
Recall [0.7069047619047619, 0.767857142857143, 0.8284523809523809,
0.7905952380952381, 0.7778571428571428]
F1 [0.6910047655090757, 0.7775030637697491, 0.8421309802698845,
0.800223457561472, 0.7847395301195276]
```

```
[ ]: hyper_params = [[8,10,0.850],[8,10,0.875],[8,10,0.9]]
     for params in hyper_params:
        ␣
      ↪random_forest_kfolds(park_fold,attr_list,park_attr,park_vals,params[0],params[1],params[2])
```

```
max_depth: 8 min_size_split: 10 maj prop: 0.85
Accuracy:  [0.7833625730994151, 0.8708771929824562, 0.8867251461988307,
0.9125146198830411, 0.881140350877193]
Precision [0.7170487115223956, 0.8530206762192056, 0.8970892304290137,
0.9156746646026832, 0.8660282446311858]
Recall [0.6527380952380952, 0.7767857142857142, 0.8095238095238095,
0.8486904761904762, 0.7970238095238096]
F1 [0.6483889547774258, 0.795804327840618, 0.8214024056971695,
0.8622904428032265, 0.8117369156767099]
max_depth: 8 min_size_split: 10 maj prop: 0.875
Accuracy:  [0.8191812865497076, 0.8969883040935673, 0.9016959064327486,
0.8966959064327487, 0.8914035087719299]
Precision [0.7662530525030525, 0.8972812971342382, 0.8844310224089635,
0.8890184407096171, 0.8598815359477125]
Recall [0.7471428571428572, 0.8228571428571427, 0.8328571428571427,
0.8161904761904761, 0.825952380952381]
F1 [0.7419883754032808, 0.8447574535679374, 0.8496829921252891,
0.8389053744580719, 0.8343172210879386]
max_depth: 8 min_size_split: 10 maj prop: 0.9
Accuracy:  [0.8292397660818713, 0.8969883040935672, 0.9066959064327487,
0.9125438596491229, 0.9066959064327487]
Precision [0.7770876427494076, 0.886200980392157, 0.907421218487395,
0.9079594017094017, 0.887734593837535]
Recall [0.7739285714285715, 0.8382142857142856, 0.836190476190476,
0.8486904761904762, 0.8517857142857144]
F1 [0.7695756301026045, 0.8525155115709898, 0.8562213917740893,
0.8666438739853645, 0.8629413142629216]
```

2.4 After analyzing the performance of each algorithm under different hyper-parameters, identify the best hyper-parameter setting

> From testing, using the best accuracy we could achieve was with max depth of 8, minimum size split of 10, and majority prop value of 0.875, ntree value of 30

2.5 For each dataset, and considering the best hyper-parameter setting for each selected algorithm, construct relevant learning curves and/or graphs

```
[ ]: hyper_params = [[8,10,0.875]]
     n_acc = []
     n_prec = []
     n_rec = []
     n_f1 = []
     for params in hyper_params:
```

```
    results =␣
↪random_forest_kfolds(park_fold,attr_list,park_attr,park_vals,params[0],params[1],params[2],
↪True)
    n_acc = results[0]
    n_prec = results[1]
    n_rec = results[2]
    n_f1 =results[3]



nvals = [1,10,20,30,40]
plt.title('Performance vs Ntrees for Parkinsons Data')
plt.xlabel('Number of Trees')
plt.ylabel('Performance')
plt.plot(nvals,n_acc,label='Accuracy',marker='o')
plt.plot(nvals,n_prec,label='Precision',marker='o')
plt.plot(nvals,n_rec,label='Recall',marker='o')
plt.plot(nvals,n_f1,label='Macro F1',marker='o')
plt.legend()
plt.text(5,0.84,'max_depth=8, min_size_split=10, maj_prop=0.875')
plt.show()
```