Programming Assignment

Raghav Juyal
EP20BTECH11018

Imports

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Part 1

Data generation

(a) Use Gaussian distribution with appropriate parameters and produce a dataset with four classes and 30 samples per class: the classes must live in the 2D space and be centered on the corners of the unit square (0,0), (0,1) (1,1), (1,0), all with independent components each with variance 0.3.
(b) Obtain a 2-class train set [X, Y] by having data on opposite corners sharing the same class with labels +1 and -1.
(c) Generate a test set [Xte , Yte] from the same distribution, starting with 200 samples per class.
(d) Visualize both sets using a scatter plot on a 2-D plane.
(e) Repeat (a)-(d) for Laplace distribution.

(a)

In [ ]:
```
# (a) 4 classes
np.random.seed(0)
centers = [[0,0],[0,1],[1,0],[1,1]]
variance = 0.3
number_of_samples = 30
classes = []

for center in centers:
    classes.append(np.random.normal(center,variance,(number_of_samples,2)))
```

(b)

In [ ]:
```
# (b) 2-class train set: (0,0) and (1,1) have label 1 and (0,1) and (1,0) have l
Ytemp = []
count = 0
Ytemp.append([1]*number_of_samples)
```

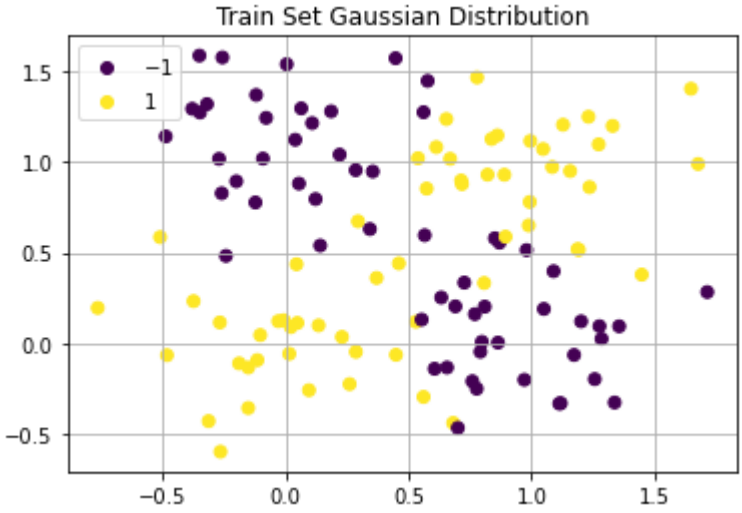
```
Ytemp.append([-1]*number_of_samples)
Ytemp.append([-1]*number_of_samples)
Ytemp.append([1]*number_of_samples)

X = np.concatenate((classes[0],classes[1],classes[2],classes[3]))
Y = np.concatenate((Ytemp[0],Ytemp[1],Ytemp[2],Ytemp[3]))
```

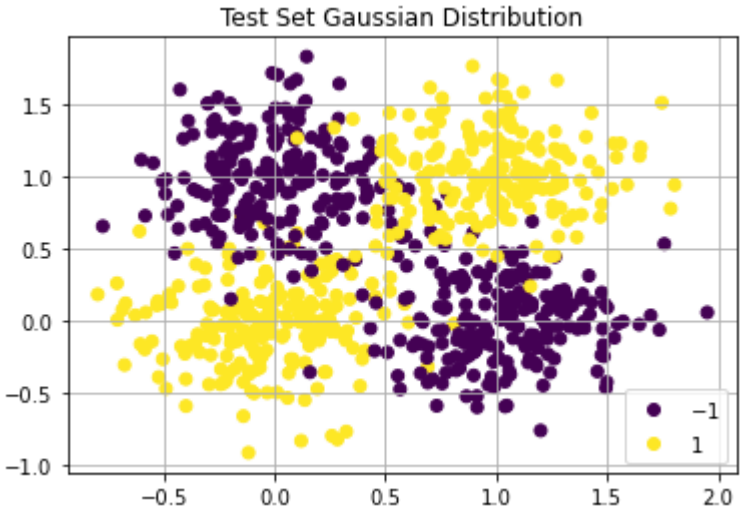
(c)

In [ ]:
```
# (c) Test Set
centers = [[0,0],[0,1],[1,0],[1,1]]
variance = 0.3
number_of_samples = 200
classes = []

for center in centers:
    classes.append(np.random.normal(center,variance,(number_of_samples,2)))

Ytemp = []
count = 0
Ytemp.append([1]*number_of_samples)
Ytemp.append([-1]*number_of_samples)
Ytemp.append([-1]*number_of_samples)
Ytemp.append([1]*number_of_samples)

Xte = np.concatenate((classes[0],classes[1],classes[2],classes[3]))
Yte = np.concatenate((Ytemp[0],Ytemp[1],Ytemp[2],Ytemp[3]))
```

(d)

In [ ]:
```
# (d) Scatter Plots
plt.grid()
plt.title("Train Set Gaussian Distribution")
scatter = plt.scatter(X[:,0],X[:,1],c=Y)
plt.legend(*scatter.legend_elements())
plt.show()
plt.grid()
plt.title("Test Set Gaussian Distribution")
scatter = plt.scatter(Xte[:,0],Xte[:,1],c=Yte)
plt.legend(*scatter.legend_elements())
plt.show()
```





(e) Repeat for Laplacian

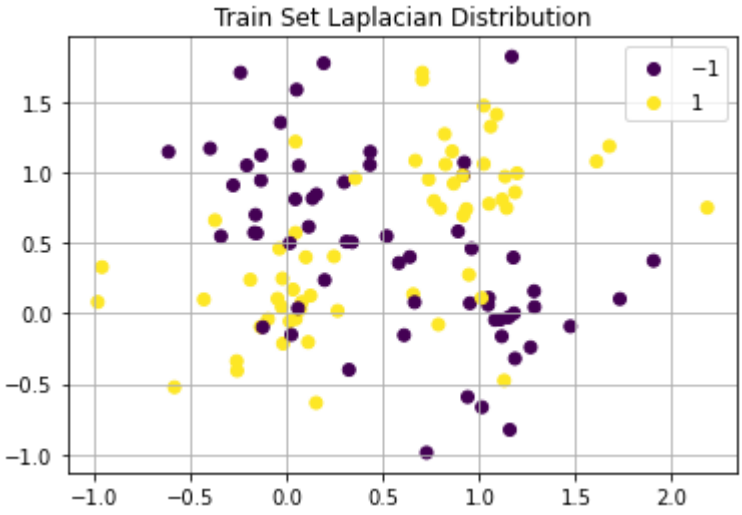
In [ ]:
```
np.random.seed(0)
centers = [[0,0],[0,1],[1,0],[1,1]]
variance = 0.3
number_of_samples = 30
classes = []

for center in centers:
    classes.append(np.random.laplace(center,variance,(number_of_samples,2)))
```

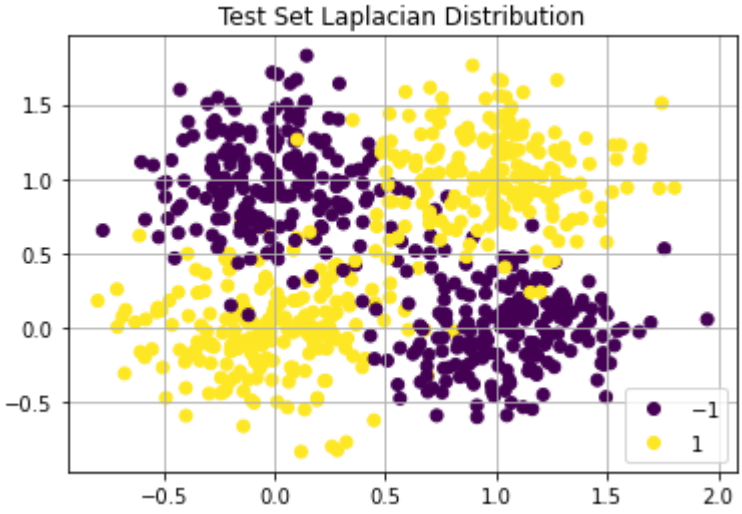
In [ ]:
```
Ytemp = []
count = 0
Ytemp.append([1]*number_of_samples)
Ytemp.append([-1]*number_of_samples)
Ytemp.append([-1]*number_of_samples)
Ytemp.append([1]*number_of_samples)

XLaplace = np.concatenate((classes[0],classes[1],classes[2],classes[3]))
YLaplace = np.concatenate((Ytemp[0],Ytemp[1],Ytemp[2],Ytemp[3]))
```

In [ ]:
```
centers = [[0,0],[0,1],[1,0],[1,1]]
variance = 0.3
number_of_samples = 200
classes = []
```

```
for center in centers:
    classes.append(np.random.normal(center,variance,(number_of_samples,2)))

Ytemp = []
count = 0
Ytemp.append([1]*number_of_samples)
Ytemp.append([-1]*number_of_samples)
Ytemp.append([-1]*number_of_samples)
Ytemp.append([1]*number_of_samples)

XteLaplace = np.concatenate((classes[0],classes[1],classes[2],classes[3]))
YteLaplace = np.concatenate((Ytemp[0],Ytemp[1],Ytemp[2],Ytemp[3]))
```

In [ ]:
```
plt.grid()
plt.title("Train Set Laplacian Distribution")
scatter = plt.scatter(XLaplace[:,0],XLaplace[:,1],c=YLaplace)
plt.legend(*scatter.legend_elements())
plt.show()
plt.grid()
plt.title("Test Set Laplacian Distribution")
scatter = plt.scatter(XteLaplace[:,0],XteLaplace[:,1],c=YteLaplace)
plt.legend(*scatter.legend_elements())
plt.show()
```





Part 2

KNN Classification

1. The k-Nearest Neighbors algorithm (kNN) assigns to a test point the most frequent label of its k closest examples in the training set. \

(a) Write a function kNNClassify to generate predictions Y_p for the 2-class data generated at Section 1. Pick a "reasonable" k.
(b) Evaluate the classification performance (prediction error) by comparing the predicted labels Y_p to the true labels Y_{te}.
(c) Visualize the obtained results, e.g. by plotting the wrongly classified points using different colors/markers.
(d) Write a function to generate & visualize the decision regions of the 2D plane that are associated with each class,
for a given classifier. Overlay the test points using scatter.

2. Parameter selection: What is a good value for k? So far we considered an arbitrary choice for k. \

You will now use the function hold outCVkNN for model selection
(a) Perform hold-out cross-validation by setting aside a fraction (ρ of the training set for validation.)
Note: You may use ρ = 0.3, and repeat the procedure 10 times. The hold-out procedure may be quite unstable.
• Use a large range of candidate values for k (e.g. k = 1, 3, 5..., 21). Notice odd numbers are considered to avoid ties.
• Repeat the process for 10 times using a random cross-validation set each time with a ρ = 0.3.
• Plot the training and validation errors for the different values of k.
• How would you now answer the question ”what is the best value for k”?
(b) How is the value of k affected by ρ (percentage of points held out) and number of repetitions? What does a large number of repetitions provide?
(c) Apply the model obtained by cross-validation (i.e., best k) to the test set and check if there is an improvement on the classification error over the result of (1).

1 (a)

In [ ]:
```
# 1.(a) kNNClassify
def kNNClassify(k,X_train,Y_train,X_test):
    neighbors = []
    ts = 1
    for x in X_test:
        distances = np.sqrt(np.sum((x-X_train)**2,axis=1))
        Y_sorted = [y for _, y in sorted(zip(distances, Y_train))]
        neighbors.append(Y_sorted[:k])

    for i in range(len(neighbors)):
        # print(neighbors[i])
        neighbors[i] = max(set(neighbors[i]), key=neighbors[i].count)
```

```
    return neighbors


k = 5
Y_p = kNNClassify(k,X,Y,Xte)
```

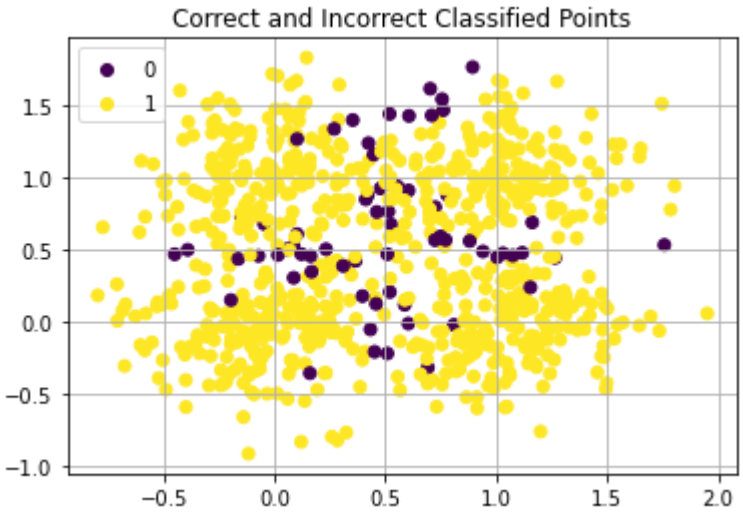
1 (b)

In [ ]:
```
# 1. (b) Classification performance
accuracy = np.count_nonzero(Yte == Y_p)/len(Yte)
print(f"Accuracy: {accuracy}")
```

```
Accuracy: 0.9175
```

1 (c)

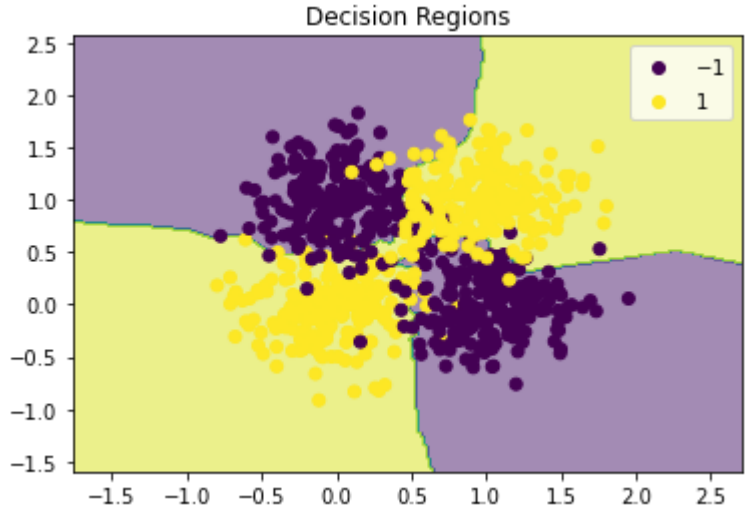
In [ ]:
```
# 1. (c) Visualization (Incorrect classified with 0 label)
plt.grid()
plt.title("Correct and Incorrect Classified Points")
scatter = plt.scatter(Xte[:,0],Xte[:,1],c=Yte==Y_p)
plt.legend(*scatter.legend_elements())
plt.show()
```



1 (d)

In [ ]:
```
# 1. (d) Decision Regions

xs, ys = np.meshgrid(np.arange(X[:, 0].min() - 1,  X[:, 0].max() + 1, 0.02), np.

Z = np.array(kNNClassify(5, X, Y, np.c_[xs.ravel(), ys.ravel()]))
Z = Z.reshape(xs.shape)

plt.contourf(xs, ys, Z, alpha=0.5)
plt.title("Decision Regions")
scatter = plt.scatter(Xte[:, 0], Xte[:, 1], c=Yte)
```

```
plt.legend(*scatter.legend_elements())
plt.show()
```



2 (a)

2. (a) \

A good value of k would be the value at which we have low validation error. This can be done by splitting our train data
into train and validation sets and doing hold out cross validation or k-fold cross validation.

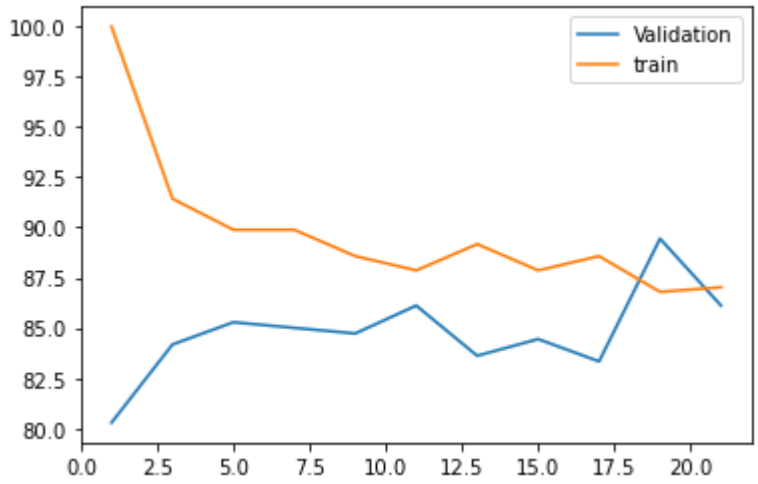
In [ ]:
```
# 2. (a)

def hold_outCVkNN(X, Y, k, n_rep, rho):
    n = len(X)
    n_hold_out = int(n*rho)
    hold_out_accuracy = np.zeros(n_rep)
    train_accuracy = np.zeros(n_rep)
    for i in range(n_rep):
        shuffle_idx = np.random.permutation(n)
        X_tr = X[shuffle_idx]
        Y_tr = Y[shuffle_idx]
        X_hold_out = X_tr[:n_hold_out]
        Y_hold_out = Y_tr[:n_hold_out]
        X_tr = X_tr[n_hold_out:]
        Y_tr = Y_tr[n_hold_out:]
        Y_pred_hold_out = kNNClassify(k,X_tr, Y_tr, X_hold_out)
        hold_out_accuracy[i] = np.count_nonzero(Y_hold_out == Y_pred_hold_out)/l
        Y_pred_train = kNNClassify(k,X_tr, Y_tr, X_tr)
        train_accuracy[i] = np.count_nonzero(Y_tr == Y_pred_train)/len(Y_tr)


    return np.round([np.mean(hold_out_accuracy)*100, np.mean(train_accuracy)*100
```

In [ ]:
```
accuracies = []
for k in range(1, 23, 2):
    acc = hold_outCVkNN(X, Y, k, 10, 0.3)
```

```
    print("k = ", k, ": ", acc)
    accuracies.append(acc)
```

```
k =  1 :  [ 80.28 100.  ]
k =  3 :  [84.17 91.43]
k =  5 :  [85.28 89.88]
k =  7 :  [85.   89.88]
k =  9 :  [84.72 88.57]
k =  11 :  [86.11 87.86]
k =  13 :  [83.61 89.17]
k =  15 :  [84.44 87.86]
k =  17 :  [83.33 88.57]
k =  19 :  [89.44 86.79]
k =  21 :  [86.11 87.02]
```

In [ ]:
```
plt.plot(range(1, 23, 2), [acc[0] for acc in accuracies], label="Validation")
plt.plot(range(1, 23, 2), [acc[1] for acc in accuracies], label="train")
plt.legend()
plt.show()
```



In [ ]:
```
print('k = ', np.argmax([x[0] for x in accuracies])*2+1, 'is the best k ')
```

```
k =  19 is the best k
```

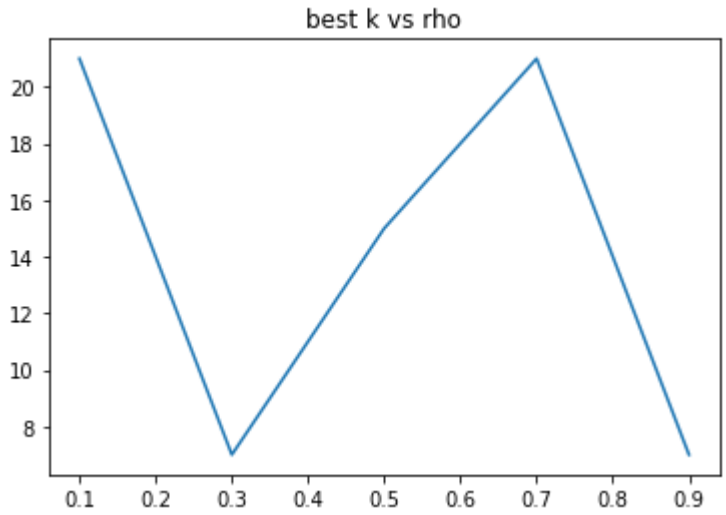
As we can see from the above graph, at k = 19, we have the largest hold out/ validation accuracy. This is why it is the best k for this problem.

2 (b)

In [ ]:
```
# For various rho
best_k = []
for rho in [0.1, 0.3, 0.5, 0.7, 0.9]:
    accuracies = []
    for k in range(1, 23, 2):
        acc = hold_outCVkNN(X, Y, k, 10, 0.3)
        accuracies.append(acc)
    best_k.append(np.argmax([x[0] for x in accuracies])*2+1)
    print('Rho = ' + str(rho) + ': best k = ', best_k[-1])
```

```
Rho = 0.1: best k =  21
Rho = 0.3: best k =  7
Rho = 0.5: best k =  15
Rho = 0.7: best k =  21
Rho = 0.9: best k =  7
```

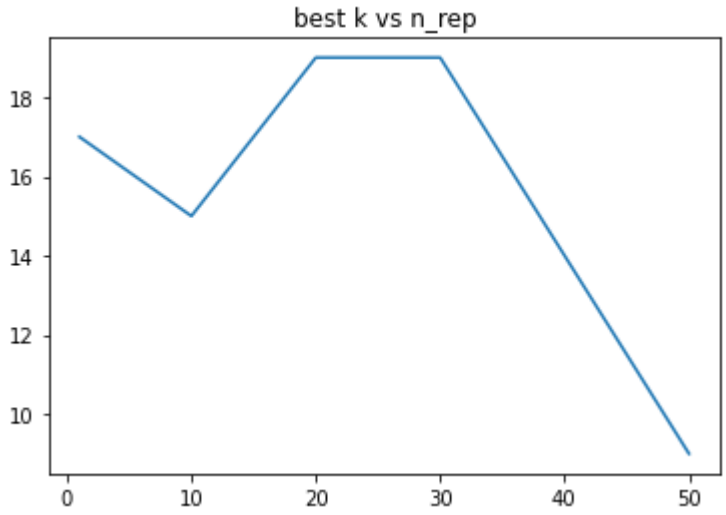
In [ ]:
```
plt.title("best k vs rho")
plt.plot([0.1, 0.3, 0.5, 0.7, 0.9], best_k)
plt.show()
```



In [ ]:
```
best_k = []
for n_rep in [1, 10, 20, 30, 50]:
    accuracies = []
    for k in range(1, 23, 2):
        acc = hold_outCVkNN(X, Y, k, 10, 0.3)
        accuracies.append(acc)
    best_k.append(np.argmax([x[0] for x in accuracies])*2+1)
    print('N_rep = ' + str(n_rep) + ': best k = ', best_k[-1])
```

```
N_rep = 1: best k =  17
N_rep = 10: best k =  15
N_rep = 20: best k =  19
N_rep = 30: best k =  19
N_rep = 50: best k =  9
```

In [ ]:
```
plt.title("best k vs n_rep")
plt.plot([1, 10, 20, 30, 50], best_k)
plt.show()
```



From the plots we see how k varies with rho and n_rep.

2 (c)

In [ ]:
```
# We saw that best k was k = 19 [for rho = 0.3 and n_rep = 10]
Y_p = kNNClassify(19,X,Y,Xte)
accuracy = np.count_nonzero(Yte == Y_p)/len(Yte)
print(f"Accuracy: {accuracy}")
```

```
Accuracy: 0.91875
```

We see that accuracy for k = 19 (0.91875) is indeed greater than for original k = 5 (0.9175). So there is an improvement in accuracy (or decrease in classification error) in the case of cross validation.

Repeat for Laplacian

1 (a)

In [ ]:
```
# 1.(a) kNNClassify
def kNNClassify(k,X_train,Y_train,X_test):
    neighbors = []
    ts = 1
    for x in X_test:
        distances = np.sqrt(np.sum((x-X_train)**2,axis=1))
        Y_sorted = [y for _, y in sorted(zip(distances, Y_train))]
        neighbors.append(Y_sorted[:k])

    for i in range(len(neighbors)):
        # print(neighbors[i])
        neighbors[i] = max(set(neighbors[i]), key=neighbors[i].count)

    return neighbors
```

```
k = 5
Y_p = kNNClassify(k,XLaplace,YLaplace,XteLaplace)
```

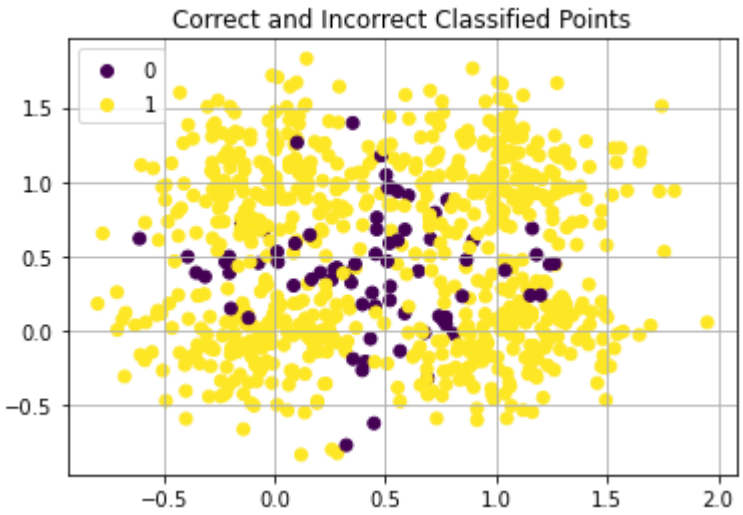
1 (b)

In [ ]:
```
# 1. (b) Classification performance
accuracy = np.count_nonzero(YteLaplace == Y_p)/len(YteLaplace)
print(f"Accuracy: {accuracy}")
```

```
Accuracy: 0.905
```

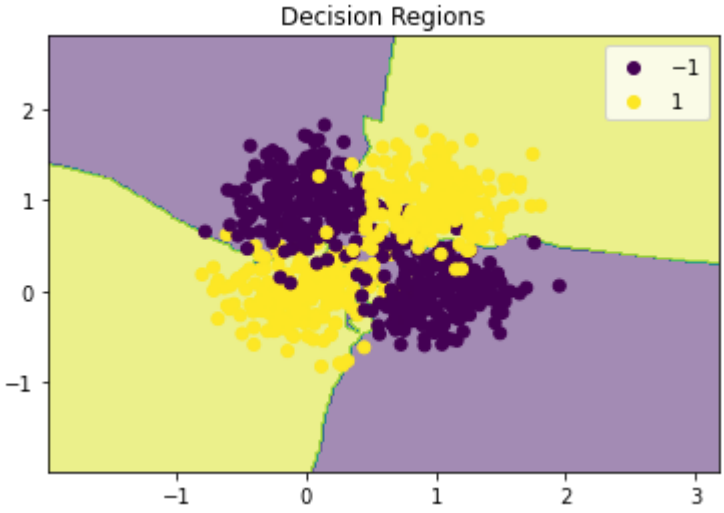
1 (c)

In [ ]:
```
# 1. (c) Visualization (Incorrect classified with 0 label)
plt.grid()
plt.title("Correct and Incorrect Classified Points")
scatter = plt.scatter(XteLaplace[:,0],XteLaplace[:,1],c=YteLaplace==Y_p)
plt.legend(*scatter.legend_elements())
plt.show()
```



1 (d)

In [ ]:
```
# 1. (d) Decision Regions

xs, ys = np.meshgrid(np.arange(XLaplace[:, 0].min() - 1,  XLaplace[:, 0].max() +

Z = np.array(kNNClassify(5, XLaplace, YLaplace, np.c_[xs.ravel(), ys.ravel()]))
Z = Z.reshape(xs.shape)

plt.contourf(xs, ys, Z, alpha=0.5)
plt.title("Decision Regions")
scatter = plt.scatter(XteLaplace[:, 0], XteLaplace[:, 1], c=YteLaplace)
plt.legend(*scatter.legend_elements())
plt.show()
```



2 (a)

2. (a) \

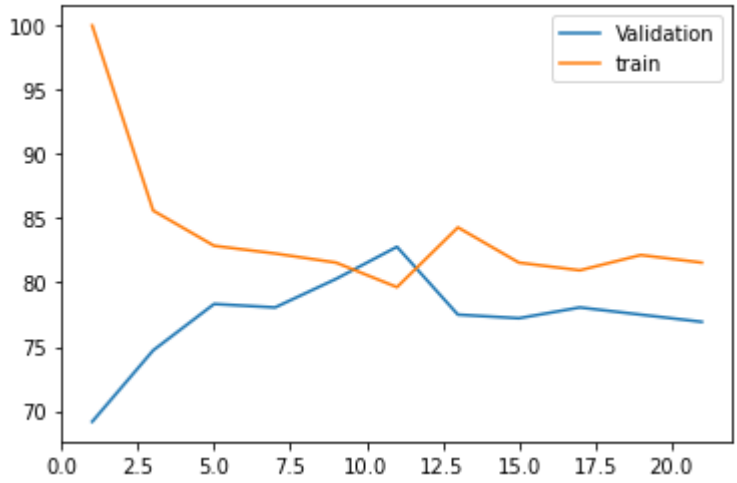
A good value of k would be the value at which we have low validation error. This can be done by splitting our train data
into train and validation sets and doing hold out cross validation or k-fold cross validation.

In [ ]:
```
# 2. (a)

def hold_outCVkNN(X, Y, k, n_rep, rho):
    n = len(X)
    n_hold_out = int(n*rho)
    hold_out_accuracy = np.zeros(n_rep)
    train_accuracy = np.zeros(n_rep)
    for i in range(n_rep):
        shuffle_idx = np.random.permutation(n)
        X_tr = X[shuffle_idx]
        Y_tr = Y[shuffle_idx]
        X_hold_out = X_tr[:n_hold_out]
        Y_hold_out = Y_tr[:n_hold_out]
        X_tr = X_tr[n_hold_out:]
        Y_tr = Y_tr[n_hold_out:]
        Y_pred_hold_out = kNNClassify(k,X_tr, Y_tr, X_hold_out)
        hold_out_accuracy[i] = np.count_nonzero(Y_hold_out == Y_pred_hold_out)/l
        Y_pred_train = kNNClassify(k,X_tr, Y_tr, X_tr)
        train_accuracy[i] = np.count_nonzero(Y_tr == Y_pred_train)/len(Y_tr)


    return np.round([np.mean(hold_out_accuracy)*100, np.mean(train_accuracy)*100
```

In [ ]:
```
accuracies = []
for k in range(1, 23, 2):
    acc = hold_outCVkNN(XLaplace, YLaplace, k, 10, 0.3)
    print("k = ", k, ": ", acc)
    accuracies.append(acc)
```

```
k =  1 :  [ 69.17 100.  ]
k =  3 :  [74.72 85.6 ]
k =  5 :  [78.33 82.86]
k =  7 :  [78.06 82.26]
k =  9 :  [80.28 81.55]
k =  11 :  [82.78 79.64]
k =  13 :  [77.5  84.29]
k =  15 :  [77.22 81.55]
k =  17 :  [78.06 80.95]
k =  19 :  [77.5  82.14]
k =  21 :  [76.94 81.55]
```

In [ ]:
```
plt.plot(range(1, 23, 2), [acc[0] for acc in accuracies], label="Validation")
plt.plot(range(1, 23, 2), [acc[1] for acc in accuracies], label="train")
plt.legend()
plt.show()
```



In [ ]:
```
print('k = ', np.argmax([x[0] for x in accuracies])*2+1, 'is the best k ')
```

```
k =  11 is the best k
```

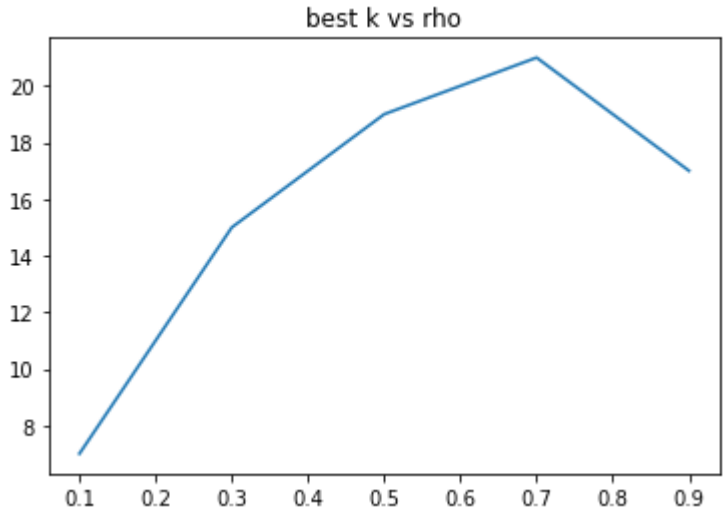
As we can see from the above graph, at k = 11, we have the largest hold out/ validation accuracy. This is why it is the best k for this problem.

2 (b)

In [ ]:
```
# For various rho
best_k = []
for rho in [0.1, 0.3, 0.5, 0.7, 0.9]:
    accuracies = []
    for k in range(1, 23, 2):
        acc = hold_outCVkNN(XLaplace, YLaplace, k, 10, 0.3)
        accuracies.append(acc)
    best_k.append(np.argmax([x[0] for x in accuracies])*2+1)
    print('Rho = ' + str(rho) + ': best k = ', best_k[-1])
```

```
Rho = 0.1: best k =  7
Rho = 0.3: best k =  15
Rho = 0.5: best k =  19
Rho = 0.7: best k =  21
Rho = 0.9: best k =  17
```

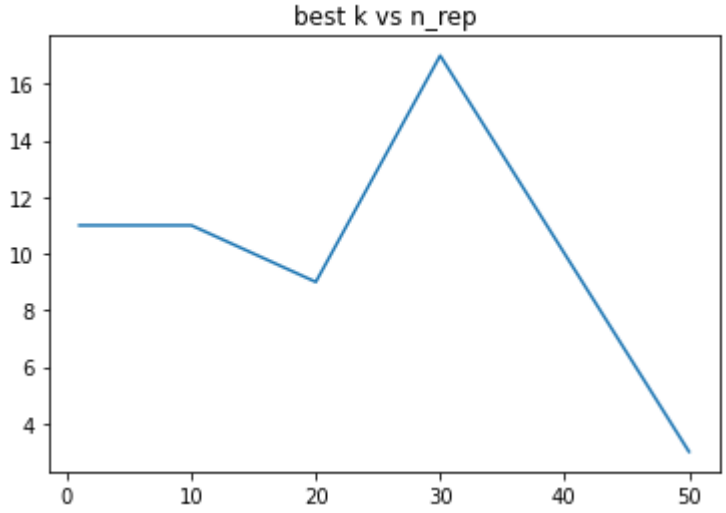
In [ ]:
```
plt.title("best k vs rho")
plt.plot([0.1, 0.3, 0.5, 0.7, 0.9], best_k)
plt.show()
```



In [ ]:
```
best_k = []
for n_rep in [1, 10, 20, 30, 50]:
    accuracies = []
    for k in range(1, 23, 2):
        acc = hold_outCVkNN(XLaplace, YLaplace, k, 10, 0.3)
        accuracies.append(acc)
    best_k.append(np.argmax([x[0] for x in accuracies])*2+1)
    print('N_rep = ' + str(n_rep) + ': best k = ', best_k[-1])
```

```
N_rep = 1: best k =  11
N_rep = 10: best k =  11
N_rep = 20: best k =  9
N_rep = 30: best k =  17
N_rep = 50: best k =  3
```

In [ ]:
```
plt.title("best k vs n_rep")
plt.plot([1, 10, 20, 30, 50], best_k)
plt.show()
```



From the plots we see how k varies with rho and n_rep.

2 (c)

In [ ]:
```
# We saw that best k was k = 11 [for rho = 0.3 and n_rep = 10]
Y_p = kNNClassify(11,XLaplace,YLaplace,XteLaplace)
accuracy = np.count_nonzero(YteLaplace == Y_p)/len(YteLaplace)
print(f"Accuracy: {accuracy}")
```

```
Accuracy: 0.90875
```

We see that accuracy for k = 11 (0.90875) is indeed greater than for original k = 5 (0.905). So there is an improvement in accuracy (or decrease in classification error) in the case of cross validation.