My ROC class with all methods accounted for:

In [ ]:
```python
class ROC():
      def __init__(self, Probs, TrueClass):
        self.Probs = Probs
        self.TrueClass = TrueClass
        self.probs_col = Probs.columns[0]
        self.tc_col = TrueClass.columns[0]


      def plot_ROC(coordinates, title):
        plt.plot(coordinates['X'], coordinates['Y'],'o--',color='r')
        plt.suptitle(title)
        plt.xlabel('X')
        plt.ylabel('Y')

        plt.show()

      def compute_AUROC(coordinates):
        sum = 0
        prev_x = 0
        for i in range(len(coordinates)):
          Xi = coordinates.iloc[i]['X']
          comp_x = Xi - prev_x
          Yi = coordinates.iloc[i]['Y']

          sum = sum + (comp_x*Yi)
          prev_x = Xi
        return sum

      def compute_ROC_coordinates(self):
        #Taking the two given dataframe objects and combining them into one dataframe


        df = self.Probs.merge(self.TrueClass, left_index=True,right_index=True)
        df = df.sort_values(by = [self.probs_col, self.tc_col], axis=0, ascending=False)
        df = df.reset_index(drop=True)

        ROC_coordinates = pd.DataFrame(columns=['X','Y'])#Creating a dataframe to store X and Y coordinates

        FP = 0
        TP = 0
        pdf = df[df[self.tc_col].str.contains('pos')]
        ndf = df[df[self.tc_col].str.contains('neg')]

        P = len(pdf)
        N = len(ndf)

        #N = (df[self.tc_col] == 'neg').sum()

        prevProb = -1 #An absurd number for the first iteration.
        prevClass = ''

        for i in range(len(df)):
          currentInstance = df.iloc[i]
          currentProb = df.iloc[i][self.probs_col]
          currentClass = df.iloc[i][self.tc_col]

          if(currentProb != prevProb or ('neg' in currentClass and 'pos' in prevClass)):
            #Calculating and adding new coordinates to the dataframe.
            x = FP/N
            y = TP/P
            coords = pd.DataFrame([[x,y]], columns=['X','Y'])
            ROC_coordinates = pd.concat([ROC_coordinates, coords])


          prevProb = currentProb
          prevClass = currentClass

          if 'pos' in currentClass:
            TP = TP + 1
          else:
            FP = FP + 1


        x_last = FP/N
        y_last = TP/P
        last_coords = pd.DataFrame([[x_last,y_last]], columns=['X','Y'])
        ROC_coordinates = pd.concat([ROC_coordinates,last_coords])

        return ROC_coordinates
```

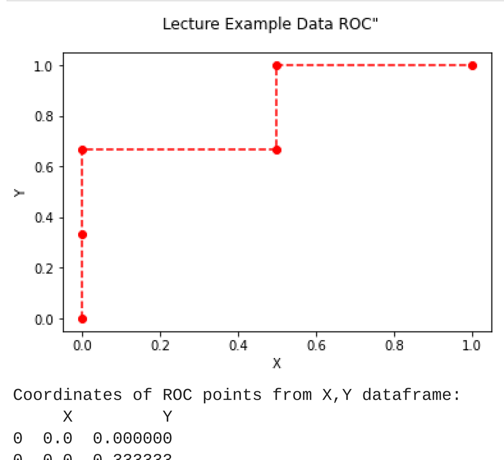
To handle duplicate probabilities in data, the old approach simply skipped adding duplicate probabilities in the coordinate list whereas I bias towards a positive result by sorting in the order of probability amount and then in the order of class, then only adding a point if the class switches from negative to positive, as this is the point in which the edges of the data lie, by doing this we create a strictly square ROC curve on any binary class dataset (might work on larger class domains) allowing for easy area under curve calculation compared to a sloped graph.

Below I have copied my kNN classifier from lab 2 and will be using k = 3 neighbors as we are going to test with the diabetes data and this was the optimal value found in that assignment:

In [ ]:
```python
# Class of k-Nearest Neigbor Classifier


class kNN():
    def __init__(self, k = 3, exp = 2):
    # constructor for kNN classifier
    # k is the number of neighbor for local class estimation
    # exp is the exponent for the Minkowski distance
        self.k = k
        self.exp = exp

    def fit(self, X_train, Y_train):
    # training k-NN method
    # X_train is the training data given with input attributes. n-th row correponds to n-th instance.
    # Y_train is the output data (output vector): n-th element of Y_train is the output value for n-th instance in X_train.
        self.X_train = X_train
        self.Y_train = Y_train

    def getDiscreteClassification(self, X_test):
    # predict-class k-NN method
    # X_test is the test data given with input attributes. Rows correpond to instances
    # Method outputs prediction vector Y_pred_test:  n-th element of Y_pred_test is the prediction for n-th instance in X_test

        Y_pred_test = [] #prediction vector Y_pred_test for all the test instances in X_test is initialized to empty list []


        for i in range(len(X_test)):   #iterate over all instances in X_test
            test_instance = X_test.iloc[i] #i-th test instance

            distances = []  #list of distances of the i-th test_instance for all the train_instance s in X_train, initially empty.

            for j in range(len(self.X_train)):  #iterate over all instances in X_train
                train_instance = self.X_train.iloc[j] #j-th training instance
                distance = self.Minkowski_distance(test_instance, train_instance) #distance between i-th test instance and j-th training instance
                distances.append(distance) #add the distance to the list of distances of the i-th test_instance

            # Store distances in a dataframe. The dataframe has the index of Y_train in order to keep the correspondence with the classes of the training instances
            df_dists = pd.DataFrame(data=distances, columns=['dist'], index = self.Y_train.index)

            # Sort distances, and only consider the k closest points in the new dataframe df_knn
            df_nn = df_dists.sort_values(by=['dist'], axis=0)
            df_knn =  df_nn[:self.k]

            # Note that the index df_knn.index of df_knn contains indices in Y_train of the k-closed training instances to
            # the i-th test instance. Thus, the dataframe self.Y_train[df_knn.index] contains the classes of those k-closed
            # training instances. Method value_counts() computes the counts (number of occurencies) for each class in
            # self.Y_train[df_knn.index] in dataframe predictions.
            predictions = self.Y_train[df_knn.index].value_counts()

            # the first element of the index predictions.index contains the class with the highest count; i.e. the prediction y_pred_test.
            y_pred_test = predictions.index[0]

            # add the prediction y_pred_test to the prediction vector Y_pred_test for all the test instances in X_test
            Y_pred_test.append(y_pred_test)

        return Y_pred_test

    def getPrediction(self, X_test):
      #This method reuses the code from getDiscreteClassification and instead of predicting a classification, returns a dataframe containing the regression values for each instance of X_test
      #instance in the X_test data's corresponding output from the Y/Class output list

        Y_regression = pd.DataFrame(columns=['regression_values']) #prediction vector Y_pred_test for all the test instances in X_test is initialized to empty list []


        for i in range(len(X_test)):   #iterate over all instances in X_test
            test_instance = X_test.iloc[i] #i-th test instance

            distances =  [] #list of distances of the i-th test_instance for all the train_instance s in X_train, initially empty.

            for j in range(len(self.X_train)):
                train_instance = self.X_train.iloc[j] #j-th training instance
                distance = self.Minkowski_distance(test_instance, train_instance)
                distances.append(distance)

            df_dists = pd.DataFrame(data=distances, columns=['dist'], index = self.Y_train.index)

            df_nn = df_dists.sort_values(by=['dist'], axis=0)
            df_knn =  df_nn[:self.k]


            predicted_values = self.Y_train[df_knn.index]
            #var = {'regression_values' : predicted_values.sum()/self.k}
            #Y_regression = Y_regression.append(var, ignore_index=True)
            Y_regression = Y_regression.append({'regression_values' : predicted_values.sum()/self.k}, ignore_index=True)

        return Y_regression

    def getClassProbs(self, X_test):

      #This method reuses the code from getDiscreteClassification and instead of predicting a classification, returns a dataframe containing the posterior probabilities of each
      #instance in the X_test data's corresponding output from the Y/Class output list

        Y_pred_test = pd.DataFrame(columns = self.Y_train.unique()) #prediction vector Y_pred_test for all the test instances in X_test is initialized to empty list []


        for i in range(len(X_test)):   #iterate over all instances in X_test
            test_instance = X_test.iloc[i] #i-th test instance

            distances =  [] #list of distances of the i-th test_instance for all the train_instance s in X_train, initially empty.

            for j in range(len(self.X_train)):  #iterate over all instances in X_train
                train_instance = self.X_train.iloc[j] #j-th training instance
                distance = self.Minkowski_distance(test_instance, train_instance) #distance between i-th test instance and j-th training instance
                distances.append(distance) #add the distance to the list of distances of the i-th test_instance

            # Store distances in a dataframe. The dataframe has the index of Y_train in order to keep the correspondence with the classes of the training instances
            df_dists = pd.DataFrame(data=distances, columns=['dist'], index = self.Y_train.index)

            # Sort distances, and only consider the k closest points in the new dataframe df_knn
            df_nn = df_dists.sort_values(by=['dist'], axis=0)
            df_knn =  df_nn[:self.k]

            # Note that the index df_knn.index of df_knn contains indices in Y_train of the k-closed training instances to
            # the i-th test instance. Thus, the dataframe self.Y_train[df_knn.index] contains the classes of those k-closed
            # training instances. Method value_counts() computes the counts (number of occurencies) for each class in
            # self.Y_train[df_knn.index] in dataframe predictions.
            class_counts= self.Y_train[df_knn.index].value_counts()

            class_values = self.Y_train.unique()
            posterior_probabilities = {}#Here we create a dictionary to append all of our probabilities to this instance as that was the most convenient way i could find to do it.

            for class_n in class_values:
              if class_n in class_counts.index:
                posterior_probabilities[class_n] = class_counts[class_n]/class_counts.sum()
              else:
                posterior_probabilities[class_n] = 0

            Y_pred_test = Y_pred_test.append(posterior_probabilities, ignore_index=True)

        return Y_pred_test


    def Minkowski_distance(self, x1, x2):
    # computes the Minkowski distance of x1 and x2 for two labeled instances (x1,y1) and (x2,y2)

        # Set initial distance to 0
        distance = 0

        # Calculate Minkowski distance using the exponent exp
        for i in range(len(x1)):
            distance = distance + abs(x1[i] - x2[i])**self.exp

        distance = distance**(1/self.exp)

        return distance

    def normalize(self, X_test): #My normalize method normalizes the training set and then uses the same min and max from the training set to normalize the test set, as that after days is
                                 #of searching I have found is the most correct method (TA approved).
      min = self.X_train.min()
      max = self.X_train.max()

      self.X_train = (self.X_train-min)/(max-min)
      X_test = (X_test-min)/(max-min)
      return self.X_train, X_test
```

Here is a block testing the ROC class and showing that all methods work correctly on a simple example. I used data from the lecture on ROC curves to create this example:

In [ ]:
```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn import tree
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from numpy.random import random
from sklearn.metrics import accuracy_score


data = pd.read_csv('/content/drive/MyDrive/ML Lab Files/Lab2/diabetes.csv')


Probs = pd.DataFrame({'Probs' : [0.99,.98,.7,.6,.43]})
TrueClass = pd.DataFrame({'TrueClass' : ['pos','pos','neg','pos','neg']})

roc = ROC(Probs, TrueClass)
coordinates = roc.compute_ROC_coordinates()
ROC.plot_ROC(coordinates, 'Lecture Example Data ROC"')
print("Coordinates of ROC points from X,Y dataframe: ")
print(coordinates)
print("Area under ROC curve: " + str(ROC.compute_AUROC(coordinates)))
```




```
Coordinates of ROC points from X,Y dataframe: 
     X         Y
0  0.0  0.000000
0  0.0  0.333333
0  0.0  0.666667
0  0.5  0.666667
0  0.5  1.000000
0  1.0  1.000000
Area under ROC curve: 0.8333333333333333
```

Below I will implement the ROC Class on the diabetes data using the kNN classifier:

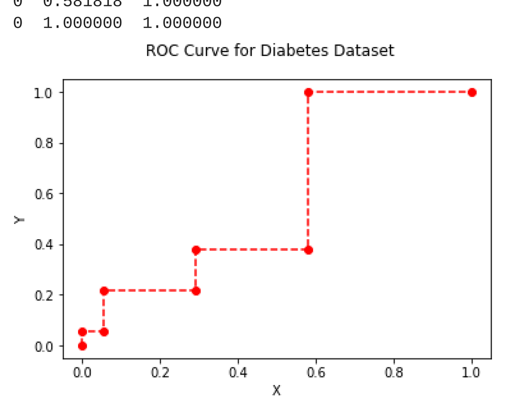
In [ ]:
```python
data_diabetes = pd.read_csv('/content/drive/MyDrive/ML Lab Files/Lab2/diabetes.csv')

Y = data_diabetes['class']
X = data_diabetes.drop(['class'],axis=1)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.34, random_state=10)

k = 3

clf = kNN(k)

clf.fit(X_train, Y_train)

class_probs_diabetes = clf.getClassProbs(X_test)
```

In [ ]:
```python
TrueClass_d = pd.DataFrame({'TrueClass' : Y_test})
Probs_d = class_probs_diabetes.drop(['tested_negative'], axis=1)

pd.set_option('display.max_rows', 1000)

diabetes_roc = ROC(Probs_d, TrueClass_d)

coordinatesdiabetes = diabetes_roc.compute_ROC_coordinates()
print(coordinatesdiabetes)
```

```
          X         Y
0  0.000000  0.000000
0  0.000000  0.054054
0  0.054545  0.054054
0  0.054545  0.216216
0  0.290909  0.216216
0  0.290909  0.378378
0  0.581818  0.378378
0  0.581818  1.000000
0  1.000000  1.000000
```

In [ ]:
```python
print(coordinatesdiabetes)

ROC.plot_ROC(coordinatesdiabetes, 'ROC Curve for Diabetes Dataset')
print("Area under curve " + str(ROC.compute_AUROC(coordinatesdiabetes)))
```

```
          X         Y
0  0.000000  0.000000
0  0.000000  0.054054
0  0.054545  0.054054
0  0.054545  0.216216
0  0.290909  0.216216
0  0.290909  0.378378
0  0.581818  0.378378
0  0.581818  1.000000
0  1.000000  1.000000
```




```
Area under curve 0.5823095823095823
```

Here we've plotted the diabetes dataset with the alternative handling of duplicate probabilities, and displayed the calculated area under the curve.