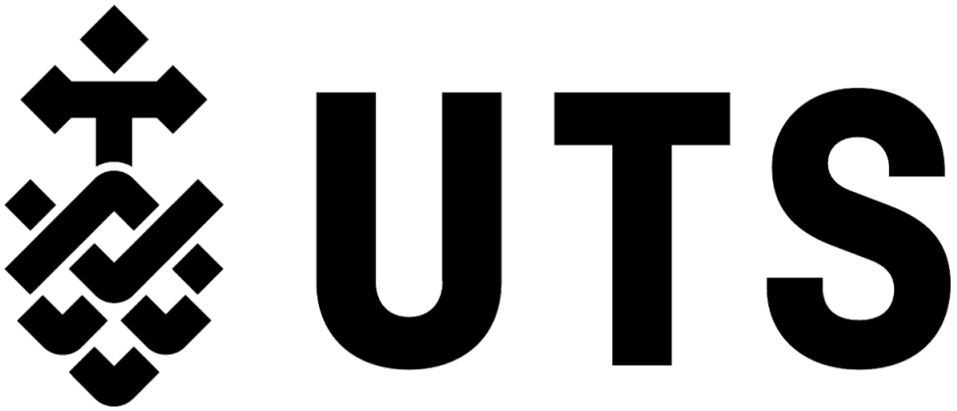

University of Technology Sydney

Andrea Bravo Avila, 24855773

September 20, 2023

Assessment Task 2: Algorithm Implementation Journal

32513 31005 Advanced Data Analytics Algorithms, Machine Learning

Contents

Introduction

The learning model that is going to be implemented is a K-nearest neighbors algorithm (KNN). This algorithm is a supervised learning method used for both regression and classification situations. This method is one that can be explained in a very geometric manner, it takes all the training data points and stores them, then when given a new data point to predict it calculates the distance between this point and all the points in the training data, then finds the k points with the least distance (nearest neighbors) and then checks what the target value of those points are, and using those predicts the value of the new data point. In its most simple form, it only needs the amount of neighbors.

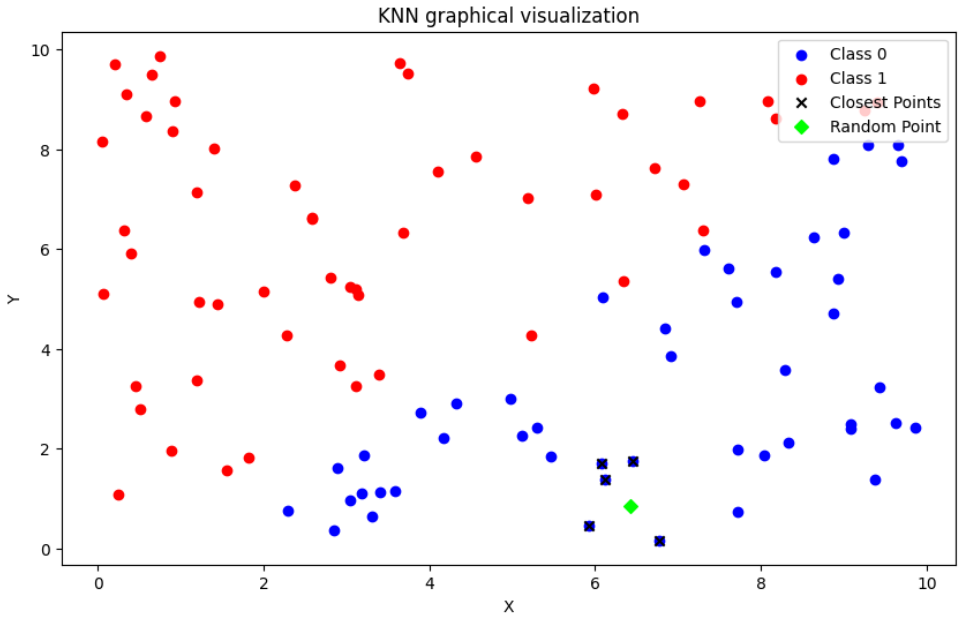

Figure 1: Visual representation of KNN

I chose the KNN algorithm because most ML algorithms work in a black-box kind of mentality, where it is easier to understand when you look at the pieces separately and assume black-boxes for them when looking at the algorithm as a whole, whereas in KNN the algorithm is very clear in what its doing at all times. In Figure 1 we can see a visual representation of what the KNN does, where we have a random dataset of points in an XY plane, that are divided into two classes, being divided by a slope. This would be the equivalent of the training dataset input into the model. Then we give it a point that we want it to classify, in the graph it is a new random point, the green diamond. It finds the closest point to it, here k=5, these five closest points are marked with an x. Then the algorithm checks the class of the five neighbors, in this case they are all class 0. Therefore the algorithm will predict that the random point will also belong to class 0.

Implementation

For the implementation of my algorithm I will be using python and creating a class called KNN that will have various parameters, methods and constraints in order to be able to create a user friendly KNN implementation.

1 Class Structure

1.1 Parameters

1.1.1 k

This parameter is how many neighbors you want the algorithm to consider as the nearest ones. The default for my implementation is 5. This parameter has a constrain to make sure that the value is a non-negative integer. When training the model, there is another constraint to make sure that k is not bigger than the amount of data points provided, when it happens it lets the user know the mistake and reverts to the default 5 and proceeding.

1.1.2 metric

One of the key parameters is what metric is being used to calculate the distance to the nearest neighbors. A metric, or a distance needs to follow specific rules in order to be valid and considered a distance, written as $d(x, y)$. The properties of a metric are:

1. **Positive**: The distance between any two points must be equal or greater to 0.
$$d(x, y) \geq 0$$

2. **Identical**: The distance between two points is 0 only if it is the same point and if it is the same point the distance must be zero.
$$d(x, y) = 0 \Leftrightarrow x = y$$

3. **Symmetrical**: The distance from x to y must be the same as the distance from y to x.
$$d(x, y) = d(y, x)$$

4. **Triangle inequality**: The distance from x to z with a middle point in y must be less or equal to the distance from x to y plus the distance from y to z.
$$d(x, z) \leq d(x, y) + d(y, z)$$

The possible metrics that can be passed as parameters in this implementation are:

- The most common metric used is **Euclidean Distance** defined by
$$d(x, y) = \sqrt{(x_1 - y_1)^2) + (x_2 - y_2)^2) + ... + (x_n - y_n)^2)}$$
This metric is the default for the implementation presented, however in high dimensional data this is not always optimal, so other types of metric can also be used to calculate the distance between the instances.

- Another way to calculate distance is the **Manhattan Distance**, which is often compared to counting blocks from place to place, since you cannot cross through the middle of buildings to get from one place to another. This metric is defined by
$$d(x, y) = |x_1 - y_1| + |x_2 - y_2| + ... + |x_n - y_n|$$

- When the data is texts and documents and we are looking for similarity in these high-dimensional spaces, another metric used is **Cosine Similarity Distance**. This measures the cosine angle between two vectors, defined by
$$d(x, y) = \frac{x \cdot y}{||x||||y||}$$

- When data is multivariate, we can find the covariance matrix, which shows the relation between each pain of variables and use this matrix to find the distance between point with the **Mahalanobis Distance**. This distance is defined by
$$d(x, y) = \sqrt{(x - y)^\top * CovarianceMatrix^{-1} * (x - y)}$$
This is useful when the data is not uniformly distributed or where the scale of the variables varies significantly.

- If the user wants to use another specific type of metric, this parameter also accepts a **Callable Function**; meaning that an externally defined function can be used instead of the preset functions available in the code. A correct function, that takes two points must be defined by the user

1.1.3 target

This parameter lets the model know if it is going to be a classification model or a regression model. When a classification model is chosen after the nearest neighbors are chosen, then it is going to predict the most common value amongst the neighbors, and it calculates the accuracy with a simple correct predictions/total prediction. When a regression model is chosen, the value assigned is the average value of the nearest neighbors, this is by default not a weighted average. To calculate the accuracy of the model, it uses the parameter, deviation 1.1.4, to see if the value predicted was within the accepted range of the real value.

1.1.4 dev

This parameter is only needed when the model is for regression, if defined when a classification model is created, a warning will appear. When using a regression model this parameter is used to define what is a correct prediction and what is an incorrect prediction, giving it an accepted error, meaning predicted y is correct if $y_{test} = y_{pred} \pm dev$. This parameter must be a non negative integer. When no value is given for a regression model, the default is set to 0, and later in the prediction process, the standard deviation of the nearest k values is used as the allowed deviation.

1.2 Methods

1.2.1 __init__(self, k=5, metric='euclidean',target='classification',dev=0)

The init method the one used to create an object of the class, the naming of this function needs to be like this in order for python to understand that it needs to create an object of the class. The parameters of this methods are the parameters of the whole class, as previously defined in section 1.1. The init method is defined with the default values, but when called, the defaults can be changed. In this method are also the constraint checks specified in the parameters.

1.2.2 fit(self, X, y)

This function is the one that trains the model, by machine learning standards the method is called fit(), and since it is a supervised type of algorithm both x and y values are passed. As explained, there is not training in KNN, but rather just saving all data points to compare to when predicting. In this section, we also find the covariance matrix if the specified metric of the KNN is mahalanobis. The fail safe for the max value of k is also added in the fit function, as it is the first approach to the data.

1.2.3 predict(self, X)

This method is called to get the predicted values of a group of data points, it returns an array with all the predicted values. To make the predictions, it calls the function make_prediction (1.2.4) for each element of the array given to it.

1.2.4 make_prediction(self,x)

This method is the main method of the KNN algorithm, it takes any given point and calculates its distance to all other points saved in the fit method. To calculate the distance it calls another function based on the specified metric of the object. From this distances, it sorts from closest to furthest, and saves the first K points. These are saved as the indexes, then it translates which indexes correspond to each point and finds the y value (or target value). Finally once it has the target values of all the k nearest neighbors, depending if its a classification model, or a regression model it will assign a target value to the new point. For classification models it will assign the most common value, for regression it will assign a simple average of the nearest values, it also calculates the standard deviation of the nearest values, when no deviation is given. This method returns a single value.

1.2.5 evaluate(self, y_pred, y_test, eval='accuracy', custom$_dev = None$)

The evalute method is meant to be used as a way to evaluate the success of the algorithm. It depends on whether the model was for regression or for classification, depending on which it will call a different external function, if it is a classification model it returns in general the result of $accuracy = \frac{Correct\ predition}{Total\ predictions} * 100$. This method returns a percentage value. If the target is regression, then the parameter eval is taken into consideration, if it is left to the default, it returns the accuracy as before, with an allowed error defined by the dev parameter; if custom_dev is not None, then the dev parameter can be changes to a new value, following the same restrictions specified for the dev parameter. This method returns a percentage value. If eval is set to score, then it returns the mean squared error.

1.2.6 cross_validation(self, X, y, folds=5, eval='accuracy)

The cross_validation method is used to evaluate how the model is performing in a more robost manner, taking into consideration that the data split into test and train sets can bias the results. This method takes the whole data set, and divides it into separate groups, and returns an average of the final evaluation for all the subsets. It takes as parameters the amount of folds, or subsets it needs to divide the dataset into, and which evaluation method is going to be used.

2 Initial implementation code

```
    class KNN:

  def __init__(self, k=5, metric='euclidean',target='classification',dev=0):
    #Parameter fail safes
    if not isinstance(k, int) or k <0:
      raise ValueError('k must be a non-negative integer')
    possible_metrics={'euclidean', 'manhattan','mahalanobis', 'cosine similarity'}
    if metric not in possible_metrics:
      raise ValueError('Invalid metric. Allowed metrics are \'euclidean\' , \'manhattan\',\'
    mahalanobis\'or \'cosine similarity\'')
    possible_targets={'classification', 'regression'}
    if target not in possible_targets:
      raise ValueError('Invalid metric. Allowed metrics are \'classification\' or \'regression
    \'')
    if not isinstance(dev, int) or dev <0:
      raise ValueError('Standard deviation must be a non-negative integer')
    self.k=k #parameter amount of neighbors
    self.metric=metric #parameter type of metric
    self.target=target #parameter target (classification or regression)
    self.dev=dev #parameter deviation for accepted deviation in regression models


  def fit(self, X, y): #fit in KNN stores the data to compare the given point
    self.X_train=X
    self.y_train=y
    self.cov_inv=cov_inv = np.linalg.inv(np.cov(X, rowvar=False))


  def predict(self, X): # calls make prediction method for each point of X
    y_pred=[self.make_prediction(x) for x in X]
    return np.array(y_pred)


  def make_prediction(self, x):
    #calculates the distance depending on the metric chosen
    distances=[calculate_distance(x, x_train, self.metric, self.cov_inv) for x_train in self.
    X_train]
    #sort distance, returns indexes of the closest k
    nearest_index=np.argsort(distances)[:self.k]
    #supervised learning compares with true value, get value of closest
    nearest_y_value=[self.y_train[i] for i in nearest_index]
    #get y value depending on the target
    return find_target(nearest_y_value, self.target)


  def evaluate(self, y_pred, y_test): #percentage of how many the model got right
    if self.target=='classification':
      return class_accuracy(y_pred, y_test)
    if self.target=='regression':
      return reg_accuracy(y_pred, y_test, self.dev)
```

Listing 1: Class definition

```
    #Other methods that don't need to belong to the class

def euclidean_distance(x1,x2): #for euclidean distance
  return np.sqrt(np.sum((x1-x2)**2))

def manhattan_distance(x1,x2): #for manhattan distance
  return np.sum(np.abs(x1-x2))

def mahalanobis_distance(x1,x2,cov_inv): #mahalanobis distance
  #cov_inv = np.linalg.inv(np.cov(X_train, rowvar=False)) passed directly from fit so its not
    calculated each time
  return np.sqrt(np.dot(np.dot((x1-x2), cov_inv), (x1-x2)))

def cosine_similarity(x1,x2):
  return np.dot(x1,x2)/(np.linalg.norm(x1)*np.linalg.norm(x2))

def most_common(values): #for classification target
  counts={}
  for item in values:
    counts[item]=counts.get(item, 0)+1
  sorted_counts=sorted(counts.items(),key=lambda x: x[1], reverse=True)
  return sorted_counts[0][0]
```

```
22
23 def calculate_distance(x1,x2,metric,cov_inv):
24   if metric=='euclidean':
25     return euclidean_distance(x1,x2)
26   if metric=='manhattan':
27     return manhattan_distance(x1,x2)
28   if metric=='mahalanobis':
29     return mahalanobis_distance(x1,x2,cov_inv)
30   if metric=='cosine similarity':
31     return cosine_similarity(x1,x2)
32
33 def find_target(nearest_y_value,target):
34   if target=='classification':
35     return most_common(nearest_y_value)
36   if target=='regression':
37     return np.mean(nearest_y_value)
38
39 def class_accuracy(y_pred, y_test):
40     error=0
41     for i in range(len(y_pred)):
42       if y_pred[i]!=y_test[i]:
43         error=error+1
44     return (len(y_pred)-error)/len(y_pred)*100
45
46 def reg_accuracy(y_pred, y_test, dev):
47   if dev==0:
48     dev=np.std(y_test)
49   error=0
50   for i in range(len(y_pred)):
51     if abs(y_pred[i]-y_test[i])>dev:
52       error=error+1
53   return (len(y_pred)-error)/len(y_pred)*100
```

Listing 2: Other methods called

3 Final Implementation

After running many trails, and debugging the code. More features were added to make the model more robust, customizable and user friendly. AI was used to get suggestions and debug the code. Some key queries are added in the 4.1. The final code is shown next:

```
1  class KNN:
2
3    def __init__(self, k=5, metric='euclidean',target='classification', dev=None):
4      #Parameter fail safes
5      if not isinstance(k, int) or k <0:
6        raise ValueError('k must be a non-negative integer')
7      possible_metrics={'euclidean', 'manhattan','mahalanobis', 'cosine similarity'}
8      if not (isinstance(metric, str) and metric in {'euclidean', 'manhattan', 'mahalanobis', '
       cosine similarity'} or callable(metric)):
9        raise ValueError('Invalid metric. Allowed metrics are \'euclidean\' , \'manhattan\',\'
       mahalanobis\'or \'cosine similarity\'; or a callable function')
10     possible_targets={'classification', 'regression'}
11     if target not in possible_targets:
12       raise ValueError('Invalid target. Allowed metrics are \'classification\' or \'regression
       \'')
13     if (not isinstance(dev, int) or dev <0) and (dev is not None):
14       raise ValueError('Standard deviation must be a non-negative integer')
15
16
17
18     self.k=k #parameter amount of neighbors
19     self.metric=metric #parameter type of metric
20     self.target=target #parameter target (classification or regression)
21     try: #parameter deviation for accepted deviation in regression models, if it isnt
       regression this parameter doesnt exist
22       if target=='classification' and dev!=0 and (dev is not None):
23         self.dev=None
24         raise ValueError("Deviation won't be used in classification models, parameter ignored"
       )
25       else:
26         self.dev=dev
27     except ValueError as e:
28       print(e)
```

```
    self.cov_inv=0

  def fit(self, X, y): #fit in KNN stores the data to compare the given point
    self.X_train=X
    self.y_train=y
    if self.metric=='mahalanobis':
      self.cov_inv=np.linalg.inv(np.cov(X, rowvar=False))
    #check that k<than amount of data points
    if self.k>len(X):
      raise ValueError("k is bigger than the amount of data points, recreate KNN with a
    different k, or add more data")



  def predict(self, X): # calls make prediction method for each point of X
    y_pred=[self.make_prediction(x) for x in X]
    return np.array(y_pred)


  def make_prediction(self, x):
    #calculates the distance depending on the metric chosen
    distances=[calculate_distance(x, x_train, self.metric, self.cov_inv) for x_train in self.
    X_train]
    #sort distance, returns indexes of the closest k
    nearest_index=np.argsort(distances)[:self.k]
    #supervised learning compares with true value, get value of closest
    nearest_y_value=[self.y_train[i] for i in nearest_index]
    if self.target=='regression' and self.dev==None:
      self.dev=np.std(nearest_y_value)
    #get y value depending on the target
    return find_target(nearest_y_value, self.target)

  def evaluate(self, y_pred, y_test, eval='accuracy', custom_dev=None):
    if self.target == 'classification':
      try:
        if eval != 'accuracy':
            raise ValueError("eval parameter is ignored in classification models, evaluation
    method is accuracy")
      except ValueError as e:
        print(e)
      try:
        if custom_dev is not None:
          raise ValueError("Deviation won't be used in classification models, parameter
    ignored")
      except ValueError as e:
        print(e)
      return class_accuracy(y_pred, y_test)
    elif self.target == 'regression':
        if eval == 'accuracy':
            if custom_dev is not None:
              if (not isinstance(custom_dev, int) or custom_dev <0) and (custom_dev is not
    None):
                raise ValueError('Standard deviation must be a non-negative integer')
              elif custom_dev >= (max(y_test) - min(y_test)):
                raise ValueError("Deviation higher than y range, guaranteed 100% accuracy, no
    significance in results")
              self.dev=custom_dev
            elif self.dev is None:
                raise ValueError("For regression evaluation with 'accuracy', you must specify
    a non-negative deviation (dev).")
            elif self.dev >= (max(y_test) - min(y_test)):
                raise ValueError("Deviation higher than y range, guaranteed 100% accuracy, no
    significance in results")
            else:
              self.dev = self.dev
            return reg_accuracy(y_pred, y_test, self.dev)
        elif eval == 'score':
            return mean_squared_error(y_pred, y_test)
        else:
            raise ValueError("Invalid eval. Allowed evaluation methods are: 'accuracy' or '
    score'")
    else:
        raise ValueError("Invalid target. Allowed targets are 'classification' or 'regression'
    ")
```

```
  def cross_validation(self, X, y, folds=5, eval='accuracy'): #cross validation for how well
    the model does splitting the data in different ways
    shuffle_index=np.random.permutation(len(X))
    fold_size=len(X)//folds
    fold_index=[shuffle_index[i:i+fold_size] for i in range(0, len(X), fold_size)]
    metrics=[]
    for i in range(folds):
      test_indices = fold_index[i]
      train_indices = np.concatenate([fold_index[j] for j in range(folds) if j != i])
      X_train = X[train_indices]
      y_train = y[train_indices]
      X_test = X[test_indices]
      y_test = y[test_indices]
      self.fit(X_train, y_train)
      y_pred = self.predict(X_test)
      metrics.append(self.evaluate(y_pred, y_test, eval))
    return np.mean(metrics)
```

Listing 3: Class definition

```

#Other methods that don't need to belong to the class

def euclidean_distance(x1,x2): #for euclidean distance
  return np.sqrt(np.sum((x1-x2)**2))

def manhattan_distance(x1,x2): #for manhattan distance
  return np.sum(np.abs(x1-x2))

def mahalanobis_distance(x1,x2,cov_inv): #mahalanobis distance
  #cov_inv = np.linalg.inv(np.cov(X_train, rowvar=False)) passed directly from fit so its not
    calculated each time
  return np.sqrt(np.dot(np.dot((x1-x2), cov_inv), (x1-x2)))

def cosine_similarity(x1,x2):
  return np.dot(x1,x2)/(np.linalg.norm(x1)*np.linalg.norm(x2))

def most_common(values): #for classification target
  counts={}
  for item in values:
    counts[item]=counts.get(item, 0)+1
  sorted_counts=sorted(counts.items(),key=lambda x: x[1], reverse=True)
  return sorted_counts[0][0]

def calculate_distance(x1,x2,metric,cov_inv):
  if metric=='euclidean':
    return euclidean_distance(x1,x2)
  if metric=='manhattan':
    return manhattan_distance(x1,x2)
  if metric=='mahalanobis':
    return mahalanobis_distance(x1,x2,cov_inv)
  if metric=='cosine similarity':
    return cosine_similarity(x1,x2)
  if callable(metric):
    try:
      return metric(x1, x2)
    except ValueError as e:
      print('Error in callable function: ',e)


def find_target(nearest_y_value,target):
  if target=='classification':
    return most_common(nearest_y_value)
  if target=='regression':
    return np.mean(nearest_y_value)

def class_accuracy(y_pred, y_test):
    error=0
    for i in range(len(y_pred)):
      if y_pred[i]!=y_test[i]:
        error=error+1
    return (len(y_pred)-error)/len(y_pred)*100

def reg_accuracy(y_pred, y_test, dev):
  error=0
  for i in range(len(y_pred)):
```

```
    if abs(y_pred[i]-y_test[i])>dev:
      error=error+1
  return (len(y_pred)-error)/len(y_pred)*100

def mean_squared_error(y_pred, y_test):
  squared_errors = (y_test - y_pred) ** 2
  mse = np.mean(squared_errors)
  return mse
```

Listing 4: Other methods called

4 Model results

4.1 Trail runs on parameter variation and dataset variation

1. **Iris Dataset (loaded from sklearn datasets) for classification situations**

   (a) Variations in k (amount of neighbors considered)

```
knn3=KNN(k=3)
knn3.fit(X_train_c, y_train_c)
y_pred=knn3.predict(X_test_c)
print(knn3.evaluate(y_pred,y_test_c))

knn5=KNN(k=5)
knn5.fit(X_train_c, y_train_c)
y_pred=knn5.predict(X_test_c)
print(knn5.evaluate(y_pred,y_test_c))

knn10=KNN(k=10)
knn10.fit(X_train_c, y_train_c)
y_pred=knn10.predict(X_test_c)
print(knn10.evaluate(y_pred,y_test_c))

try:
  knn_not_k=KNN(k=(len(X_train_c)+1)) # giving a wrong value for the parameter
  knn_not_k.fit(X_train_c, y_train_c)
except ValueError as e:
  print(f"Exception caught: {e}")
```

```
94.73684210526315
97.36842105263158
94.73684210526315
Exception caught: k is bigger than the amount of data points, recreate KNN with a different k, or add more data
```

Figure 2: Variations in k

   (b) Variation in type of metric used

```
knn1=KNN(k=5,metric='euclidean') #euclidean distance
knn1.fit(X_train_c, y_train_c)
y_pred=knn1.predict(X_test_c)
print(knn1.evaluate(y_pred,y_test_c))

knn2=KNN(k=5,metric='manhattan') #manhattan distance
knn2.fit(X_train_c, y_train_c)
y_pred=knn2.predict(X_test_c)
print(knn2.evaluate(y_pred,y_test_c))

knn3=KNN(k=5,metric='mahalanobis') #mahalanobis distance
knn3.fit(X_train_c, y_train_c)
y_pred=knn3.predict(X_test_c)
print(knn3.evaluate(y_pred,y_test_c))

knn4=KNN(k=5,metric='cosine similarity') #cosine similarity distance
knn4.fit(X_train_c, y_train_c)
y_pred=knn4.predict(X_test_c)
print(knn4.evaluate(y_pred,y_test_c))

knn4=KNN(k=5,metric=euclidean_distance) #euclidean distance passing the external function directly as a callable function
knn4.fit(X_train_c, y_train_c)
y_pred=knn4.predict(X_test_c)
print(knn4.evaluate(y_pred,y_test_c))

try:
  knn_not_metric=KNN(metric='not metric') # giving a wrong value for the parameter
except ValueError as e:
  print(f"Exception caught: {e}")
```

```
97.36842105263158
94.73684210526315
89.47368421052632
0.0
97.36842105263158
Exception caught: Invalid metric. Allowed metrics are 'euclidean' , 'manhattan','mahalanobis'or 'cosine similarity'; or a callable function
```

Figure 3: Change in metrics parameter

   (c) Cross Validation

```
knn_cv=KNN(k=10, metric='euclidean')
knn_cv.cross_validation(X_c, y_c)

96.66666666666667
```

Figure 4: Caption

2. **Diabetes Dataset (loaded from sklearn datasets) for regression situations**

   (a) Variations in k (amount of neighbors considered)

```
[54] knn3=KNN(k=5,target='regression')
     knn3.fit(X_train_r, y_train_r)
     y_pred=knn3.predict(X_test_r)
     print(knn3.evaluate(y_pred,y_test_r))

     knn5=KNN(k=10,target='regression')
     knn5.fit(X_train_r, y_train_r)
     y_pred=knn5.predict(X_test_r)
     print(knn5.evaluate(y_pred,y_test_r))

     knn10=KNN(k=20,target='regression')
     knn10.fit(X_train_r, y_train_r)
     y_pred=knn10.predict(X_test_r)
     print(knn10.evaluate(y_pred,y_test_r))

     try:
       knn_not_k=KNN(k=(len(X_train_r)+1), target='regression') # giving a wrong value for the parameter
       knn_not_k.fit(X_train_c, y_train_c)
     except ValueError as e:
       print(f"Exception caught: {e}")

     21.62162162162162
     48.64864864864865
     67.56756756756756
     Exception caught: k is bigger than the amount of data points, recreate KNN with a different k, or add more data
```

Figure 5: Variations in k

   (b) Variation in type of metric used

```
[60] knn1=KNN(k=20,metric='euclidean',target='regression')
     knn1.fit(X_train_r, y_train_r)
     y_pred=knn1.predict(X_test_r)
     print(knn1.evaluate(y_pred,y_test_r))

     knn2=KNN(k=20,metric='manhattan',target='regression')
     knn2.fit(X_train_r, y_train_r)
     y_pred=knn2.predict(X_test_r)
     print(knn2.evaluate(y_pred,y_test_r))

     knn3=KNN(k=20,metric='mahalanobis',target='regression')
     knn3.fit(X_train_r, y_train_r)
     y_pred=knn3.predict(X_test_r)
     print(knn3.evaluate(y_pred,y_test_r))

     knn4=KNN(k=20,metric='cosine similarity',target='regression')
     knn4.fit(X_train_r, y_train_r)
     y_pred=knn4.predict(X_test_r)
     print(knn4.evaluate(y_pred,y_test_r))

     67.56756756756756
     73.87387387387388
     70.27027027027027
     36.93693693693694
```

Figure 6: Change in metrics parameter

(c) Variations in dev (allowed deviation for accuracy evaluation)

```
[61] knn1=KNN(k=20,metric='manhattan',target='regression', dev=0) #std of k nearest
     knn1.fit(X_train_r, y_train_r)
     y_pred=knn1.predict(X_test_r)
     print(knn1.evaluate(y_pred,y_test_r))

     print('\nImportant values:')
     print('Standard Deviation of k nearest in default:', knn1.dev)
     print('Range of possible y values: ',max(y_test_r)-min(y_test_r),'\n')

     knn2=KNN(k=20,metric='manhattan',target='regression', dev=30) #10% of range
     knn2.fit(X_train_r, y_train_r)
     y_pred=knn2.predict(X_test_r)
     print(knn2.evaluate(y_pred,y_test_r))

     knn3=KNN(k=20,metric='manhattan',target='regression', dev=80) # More than the std dev
     knn3.fit(X_train_r, y_train_r)
     y_pred=knn3.predict(X_test_r)
     print(knn3.evaluate(y_pred,y_test_r))

     knn4=KNN(k=20,metric='manhattan',target='regression', dev=307) #Range, guaranteed 100%
     knn4.fit(X_train_r, y_train_r)
     y_pred=knn4.predict(X_test_r)
     try:
       knn4.evaluate(y_pred,y_test_r)
     except ValueError as e:
       print(f"Exception caught: {e}")

     73.87387387387388

     Important values:
     Standard Deviation of k nearest in default: 61.30683077765479
     Range of possible y values:  315.0

     35.13513513513514
     84.68468468468468
```

Figure 7: Variations in dev

(d) Variations in eval (evaluation methods)

```
[62] knn1=KNN(k=20,metric='manhattan',target='regression')
     knn1.fit(X_train_r, y_train_r)
     y_pred=knn1.predict(X_test_r)
     print(knn1.evaluate(y_pred,y_test_r, eval='accuracy'))

     knn2=KNN(k=20,metric='manhattan',target='regression')
     knn2.fit(X_train_r, y_train_r)
     y_pred=knn2.predict(X_test_r)
     print(knn2.evaluate(y_pred,y_test_r, eval='score'))

     knn4=KNN(k=20,metric='manhattan',target='regression')
     knn4.fit(X_train_r, y_train_r)
     y_pred=knn4.predict(X_test_r)
     try:
       knn4.evaluate(y_pred,y_test_r, eval='not a method')
     except ValueError as e:
       print(f"Exception caught: {e}")

     73.87387387387388
     3111.1633558558565
     Exception caught: Invalid eval. Allowed evaluation methods are: 'accuracy' or 'score'
```

Figure 8: Variations in eval

(e) Cross validation

```
[68] knn_cv_r=KNN(k=20, metric='manhattan', target='regression')
     knn_cv_r.cross_validation(X_r, y_r, folds=10)

     73.18181818181817
```

Figure 9: Caption

ChatGPT Appendix

4.2 What are ways to calculate distance between two point in high dimentional data

When dealing with high-dimensional data, you can use various distance metrics to calculate the distance between two points. Some of the common distance metrics for high-dimensional data include:

- Euclidean Distance (L2 Norm):

  - Calculates the straight-line distance between two points in Euclidean space.

  - Formula: $d(x, y) = sqrt(sum((xi - yi)^2))$ for each dimension i.

- Manhattan Distance (L1 Norm):

  - Calculates the sum of absolute differences between the coordinates of two points.

  - Formula: $d(x, y) = sum(|xi - yi|)$ for each dimension i.

- Minkowski Distance (Lp Norm):

  - A generalization of both Euclidean and Manhattan distances.

  - You can control the power parameter p, which influences the behavior of the distance metric.

  - Formula: $d(x, y) = (sum(|xi - yi|^p))^{(1/p)}$ for each dimension i.

- Cosine Similarity:

  - Measures the cosine of the angle between two vectors.

  - Often used for text and document similarity in high-dimensional spaces.

  - Formula: $cosine\_similarity(x, y) = (x \cdot y)/(||x|| * ||y||)$, where dot represents the dot product, and $||x||$ and $||y||$ are the magnitudes of the vectors.

- Jaccard Similarity:

  - Measures the similarity between two sets.

  - Commonly used in data mining for finding similarities between sets of items.

  - Formula: $J(A, B) = |A \cap B|/|A \cup B|$, where $|A \cap B|$ is the size of the intersection of sets A and B, and $|A \cup B|$ is the size of the union of sets A and B.

- Hamming Distance:

  - Calculates the number of positions at which two binary strings of equal length differ.

  - Suitable for comparing binary data or categorical data.

  - Formula: hamming_distance(x, y) = sum(xi != yi) for each position i.

- Mahalanobis Distance: - Takes into account the correlations between dimensions.

  - Useful for dealing with multivariate data.

  - Formula: $mahalanobis(x, y) = sqrt((x - y)T * C^{-1} * (x - y))$, where C is the covariance matrix.

- Correlation Distance: - Measures the correlation between two vectors.

  - Used for comparing the similarity of patterns or trends.

  - Formula: correlation_distance(x, y) = 1 - corr(x, y), where corr(x, y) is the Pearson correlation coefficient.

The choice of distance metric depends on the nature of your data and the specific problem you are trying to solve. It's important to select a distance metric that aligns with the characteristics and goals of your analysis. In high-dimensional spaces, the "curse of dimensionality" can impact the effectiveness of some distance metrics, so careful consideration is needed. Additionally, dimensionality reduction techniques like PCA or t-SNE may be applied to mitigate these issues in high-dimensional data.

4.3 Can a knn algorithm use the mahalanobis disntace to find the neighbors? How would this impact computational power needed? when would it be useful if at all?

Here's how using the Mahalanobis distance in KNN can impact computational power and when it might be useful:

1. Impact on Computational Power:

Using the Mahalanobis distance in KNN can increase the computational complexity and memory requirements compared to using simpler distance metrics like Euclidean or Manhattan distance. This is because computing the Mahalanobis distance involves calculating the inverse of the covariance matrix, which can be computationally expensive, especially in high-dimensional spaces.

Increased Complexity: Calculating the inverse of the covariance matrix has a time complexity of $O(d^3)$, where d is the number of dimensions. This can be prohibitive in high-dimensional spaces.

Memory Requirements: Storing and manipulating the covariance matrix can also require significant memory, especially when dealing with large datasets.

Robustness: The Mahalanobis distance is sensitive to outliers and the quality of the covariance estimate. Outliers or poorly estimated covariances can lead to unreliable results.

2. Usefulness:

The usefulness of the Mahalanobis distance in KNN depends on the characteristics of your data and the specific problem you are trying to solve:

Correlated Features: If your dataset contains highly correlated features, the Mahalanobis distance can better capture the relationships between variables compared to simpler distance metrics. In such cases, it may improve the accuracy of KNN.

Variable Scales: When features have different scales, the Mahalanobis distance can help by normalizing the distances, ensuring that all dimensions contribute equally to the distance calculation.

Multivariate Data: In applications where you have multivariate data (e.g., sensor readings), and you expect correlations between sensors, the Mahalanobis distance can be more appropriate.

Robustness: In situations where you want to reduce the influence of outliers, the Mahalanobis distance can be advantageous, as it considers the covariance structure and can downweight outliers.

In summary, using the Mahalanobis distance in KNN can be beneficial when dealing with data that has correlated features, different variable scales, or when robustness to outliers is important. However, you should be aware of the increased computational complexity and memory requirements, especially in high-dimensional spaces. It's essential to assess whether the benefits in accuracy outweigh the computational costs for your specific problem.

4.4 Suggestions after Initial Implementation

The provided KNN class looks well-structured and includes parameter validation, which is a good practice. However, there are still some ways to improve the class for better functionality, flexibility, and maintainability:

Documentation: Add docstrings to your class and methods to describe their purpose, input parameters, and expected output. This will make your code more understandable for others and for yourself in the future.

Scalability: Consider handling cases where the number of neighbors (k) can be greater than the number of training data points. This can be a potential issue when k is large, and you don't have enough training data.

Support for Custom Distance Metrics: Allow users to define custom distance metrics. Instead of hardcoding specific metrics like 'euclidean', 'manhattan', 'mahalanobis', or 'cosine similarity', you can accept a callable distance function as a parameter. This would make the class more flexible.

Parameter Validation: For the 'dev' parameter, which is relevant only for regression, add checks to ensure it's only used when 'target' is 'regression' and that it's non-negative.

Standardization: Consider adding an option to standardize the input data before fitting the model. Standardization can be important for some distance metrics, like Euclidean distance.

Distance Metric Function: Ensure that the calculate_distance function you mentioned in the make_prediction method is correctly implemented or imported.

Cross-Validation: Implement methods for cross-validation, such as k-fold cross-validation, to assess the model's performance more reliably. This can be useful for both classification and regression tasks.

Model Persistence: Add methods to save and load trained models. This allows users to save their trained KNN models for later use without retraining.

Error Handling: Enhance error handling by providing informative error messages that help users understand what went wrong when they provide incorrect input.

Testing: Implement unit tests for your class to ensure that it functions correctly under various scenarios.

Efficiency: Consider optimizing your code for performance, especially for large datasets. For example, you can explore methods to reduce unnecessary computations in distance calculations.

Optional Features: Add optional features like weighting neighbors by distance or using different strategies for tie-breaking when selecting neighbors.

Code Organization: Organize your code into smaller, more focused methods to improve readability and maintainability.

By considering these suggestions and continually refining your class, you can create a more versatile and robust KNN implementation.