Exercise 20

June 28, 2022

1 Exercise 20

1.1 a)

The algorithm will almost only generate clusters based on the feature with the biggest magnitude because the euklidian distance is calculated using the difference in each feature. A solution is to scale the data, so that all attributes have similar magnitudes.

1.2 b)

The k-NN algorithm is a very simple algorithm. It is called a "lazy learner" because it does not learn some sort of decision function, but memorizes the data instead. Therefore you have no real "training time" for k-NN, but the predictions are more resource expensive.

Other classifiers like a random forest have a learning phase which needs a certain amount of time but are faster in the classification itself. Exspacially for high dimensional data with many points k-NN gets very slow, while a random forest still performs good.

1.3 c)

[1]:
```
import numpy as np
import pandas as pd

# read data
S = pd.read_hdf("NeutrinoMC.hdf5", key = "Signal")
B = pd.read_hdf("NeutrinoMC.hdf5", key = "Background")
```

[2]:
```
import statistics
from statistics import mode

class KNN:
    '''KNN Classifier.

    Attributes
    ----------
    k : int
        Number of neighbors to consider.
    '''
    def __init__(self, k):
```

```
        '''Initialization.
        Parameters are stored as member variables/attributes.

        Parameters
        ----------
        k : int
            Number of neighbors to consider.
        '''
        self.k = k

    def fit(self, X, y):
        '''Fit routine.
        Training data is stored within object.

        Parameters
        ----------
        X : numpy.array, shape=(n_samples, n_attributes)
            Training data.
        y : numpy.array shape=(n_samples)
            Training labels.
        '''
        self.X_train = X
        self.y_train = y

    def predict(self, X):
        '''Prediction routine.
        Predict class association of each sample of X.

        Parameters
        ----------
        X : numpy.array, shape=(n_samples, n_attributes)
            Data to classify.

        Returns
        -------
        prediction : numpy.array, shape=(n_samples)
            Predictions, containing the predicted label of each sample.
        '''
        prediction = [] # empty list that will include predicted labels

        for x_test in X: # loop over all test data

            distances = np.zeros(len(self.X_train)) # empty array for sorting␣
↪distances

            # to sort the distance by length we can calculate the squared␣
↪distance, since it's faster
```

```
            for i in range(len(distances)): # Calculate distance to all␣
↪training data points
                c = x_test - self.X_train[i] # connection vector (faster than␣
↪computing dot(a-b, a-b))
                distances[i] = np.dot(c, c) # no need for sqrt, since we only␣
↪sort by size

            indices = np.argsort(distances) # sort for smallest distance

            # find most frequent label for k nearest neighbors
            keys = []
            for k in range(self.k):
                keys.append(self.y_train[indices[k]])

            prediction.append(mode(keys))    # mode(list) returns most frequent␣
↪item in list

        return prediction
```

1.4 d)

[3]:
```
# getting only data with no NaN's
signals = S[["NumberOfHits", "x", "y"]][S["x"].notna() * S["y"].notna() *␣
 ↪S["NumberOfHits"].notna()]
background = B[["NumberOfHits", "x", "y"]]
events = pd.concat([signals, background])

# list of labels
labels = ["Signal"]*len(signals)
labels.extend(["Background"]*len(background))

n_signal = 10000
n_background = 20000

# training data (equal ratios of signal / background)
X_train = np.array(pd.concat([events[:2500], events[30000:32500]]))
y_train = labels[:2500]
y_train.extend(labels[30000:32500])
y_train = np.array(y_train)

# test data
X_test = np.array(pd.concat([events[5000:(5000 + n_signal)], events[40000:
 ↪(40000 + n_background)]]))
y_test = labels[5000:(5000 + n_signal)]
y_test.extend(labels[40000:(40000 + n_background)])
```

```
y_test = np.array(y_test)
```

[4]:
```
# Applying fit and prediction with k = 10
knn10 = KNN(k = 10)

knn10.fit(X_train, y_train)

result = knn10.predict(X_test)
```

[5]:
```
tp = len(y_test[(y_test == result) * (y_test == "Signal")])
tn = len(y_test[(y_test == result) * (y_test == "Background")])
fn = len(y_test[(y_test != result) * (y_test == "Background")])
fp = len(y_test[(y_test != result) * (y_test == "Signal")])

precision = tp/(tp + fp)
recall = tp/(tp + fn)
accuracy = (tp + tn)/(n_signal + n_background)

print("precision: ", precision)
print("recall   : ", recall)
print("accuracy : ", accuracy)
```

```
precision:  0.9573
recall   :  0.8282574839937705
accuracy :  0.9196
```

1.5 e)

[6]:
```
# Applying log10 on number of hits feature
X_train_log10 = X_train
X_test_log10 = X_test


X_train_log10[:,0] = np.log10(X_train_log10[:,0])
X_test_log10[:,0] = np.log10(X_test_log10[:,0])
```

[7]:
```
knn10.fit(X_train_log10, y_train)

result2 = knn10.predict(X_test_log10)
```

[8]:
```
tp = len(y_test[(y_test == result2) * (y_test == "Signal")])
tn = len(y_test[(y_test == result2) * (y_test == "Background")])
fn = len(y_test[(y_test != result2) * (y_test == "Background")])
fp = len(y_test[(y_test != result2) * (y_test == "Signal")])

precision = tp/(tp + fp)
recall = tp/(tp + fn)
```

```
accuracy = (tp + tn)/(n_signal + n_background)

print("precision: ", precision)
print("recall   : ", recall)
print("accuracy : ", accuracy)
```

```
precision:  0.9833
recall   :  0.8617123827885373
accuracy :  0.9418333333333333
```

Scaling the number of hits using a log function, changes it's magnitude to numbers closer to the x and y values and therefore gives better results, because the classification is not focused that much on the number oh hits.

1.6 f)

[9]:
```
# same k-nn with k = 20
knn20 = KNN(k = 20)

knn20.fit(X_train, y_train)

result3 = knn10.predict(X_test)
```

[10]:
```
tp = len(y_test[(y_test == result3) * (y_test == "Signal")])
tn = len(y_test[(y_test == result3) * (y_test == "Background")])
fn = len(y_test[(y_test != result3) * (y_test == "Background")])
fp = len(y_test[(y_test != result3) * (y_test == "Signal")])

precision = tp/(tp + fp)
recall = tp/(tp + fn)
accuracy = (tp + tn)/(n_signal + n_background)

print("precision: ", precision)
print("recall   : ", recall)
print("accuracy : ", accuracy)
```

```
precision:  0.9833
recall   :  0.8617123827885373
accuracy :  0.9418333333333333
```

Changing k from 10 to 20 coincidentally gives the same result as e). It is not always better to use a higher k, but in this case it seems to work good when compared to k = 10.