Rajshahi University of Engineering and Technology

Department of Computer Science and Engineering

CSE 4204

Lab Report

Submitted By:
Riyad Morshed Shoeb
Roll: 1603013
Computer Science and Engineering
Rajshahi University of Engineering and Technology

Submitted To:
Rizoan Toufiq
Assistant Professor
Computer Science and Engineering
Rajshahi University of Engineering and Technology

July 24, 2022

K Nearest Neighbor

Introduction

K-Nearest Neighbour, abbreviated as KNN, is one of the simplest Machine Learning algorithms based on Supervised Learning technique. This algorithm assumes the similarity between the new case/data and available cases and put the new case into the category that is most similar to the available categories. It stores all the available data and classifies a new data point based on the similarity. This means when new data appears, it can be easily classified into a well suite category by using the algorithm.

There are various distance matrices.

Euclidean Distance

$$d(X,Y)_{euc} = \sqrt{\left(\sum_{i=1}^{n}(X_i - Y_i)^2\right)}$$

City block Distance (Manhattan)

$$D_{eb} = \sum_{n} |X_i - Y_i|$$

Square Distance $D_{sq} = MAX|X_i - Y_i|$

The Algorithm

1. Select the number neighbors (K).
2. Calculate the distance between K number of neighbors and the rogue pattern.
3. Assign class to the rogue pattern using a discriminant function, $f(X)$. For a two-class problem:
$$f(X) = closest(class_1) - closest(class_2)$$
If $f(X)$ is positive, the pattern belongs to $class_2$, and $class_1$ otherwise.

Source Code

```python
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt

class KNearestNeighbor():
    def __init__(self, X, y):
        self.X = X
        self.y = y
    def predict(self, x, measure="euclidean"):
        closest_class_1 = self.distance(self.X[self.y==0], x, measure)
        closest_class_2 = self.distance(self.X[self.y==1], x, measure)
        f_x = closest_class_1 - closest_class_2
```

```
            return 1 if f_x>0 else 0
    def distance(self, X, x, measure="euclidean"):
        d_X_x = None
        if measure=="city-block":
            d_X_x = self._city_block(X, x)
        elif measure=="square":
            d_X_x = self._square(X, x)
        else:
            d_X_x = self._euclidean(X, x)
        return d_X_x
    def _euclidean(self, X, x):
        d = 0
        for sample in X:
            d += np.sqrt((sample[0]-x[0])**2 + (sample[1]-x[1])**2)
        d /= len(X)
        return d
    def _city_block(self, X, x):
        d = 0
        for sample in X:
            d += np.abs((sample[0]-x[0]) + (sample[1]-x[1]))
        d /= len(X)
        return d
    def _square(self, X, x):
        d = 0
        for sample in X:
            d += np.max([(sample[0]-x[0]),  (sample[1]-x[1])])
        d /= len(X)
        return d

X, y = datasets.make_blobs(n_samples=1500, n_features=2, centers=2, cluster_std=1.5)

sc = plt.scatter(X[:,0], X[:,1], c=y, marker='.')
plt.colorbar(sc)
plt.title('Classified Data')
plt.savefig('classified-input.png')
plt.show()

model = KNearestNeighbor(X, y)
x = np.random.normal(loc=0, size=2) * 5
predicted = model.predict(x)

sc = plt.scatter(X[:,0], X[:,1], c=y, marker='.')
plt.scatter(x[0], x[1], marker='D')
plt.colorbar(sc)
plt.title(f"Rogue input: {x} is classified as class {predicted}")
plt.savefig('rogue-input.png')
plt.show()
```

Output

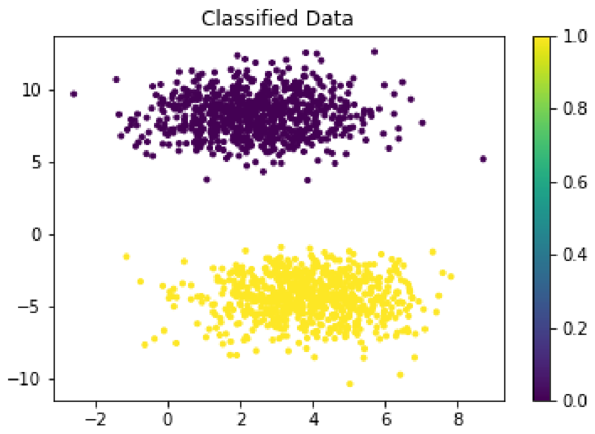

Figure 1: Classified Data

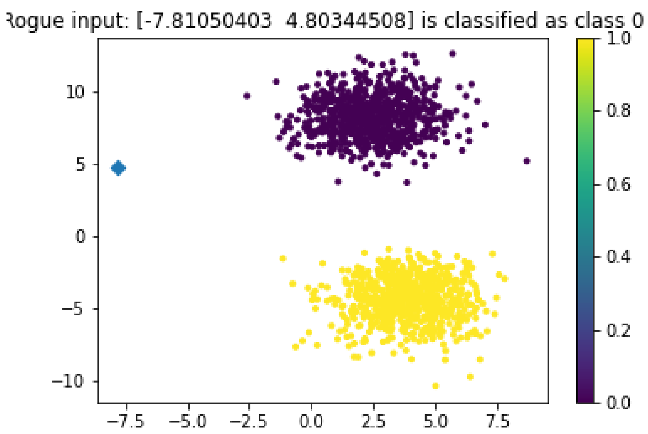

Figure 2: Classifying a rogue pattern

Advantages

1. It is simple to implement.
2. It is robust to the noisy training data.
3. It can be more effective if the training data is large.

Disadvantages

1. Always needs to determine the value of K which may be complex some time.
2. The computation cost is high because of calculating the distance between the data points for all the training samples.

Single Layer Perceptron Learning Algorithm

Introduction

The Perceptron algorithm is a two-class (binary) classification machine learning algorithm. It is a type of neural network model, perhaps the simplest type of neural network model. It consists of a single node or neuron that takes a row of data as input and predicts a class label. This is achieved by calculating the weighted sum of the inputs and a bias. The weighted sum of the input of the model is called the activation.

$$y = f_h\left(\sum_{i=1}^{n} w_i x_i - \theta\right)$$

If the activation is above 0, the model will output 1; otherwise, it will output 0.

$$\begin{aligned} f_h(x) &= 1 \qquad & x > 0 \\ f_h(x) &= 0 \qquad & x \leq 0 \end{aligned}$$

The Perceptron is a linear classification algorithm. This means that it learns a decision boundary that separates two classes using a line in the feature space.

The Algorithm

1. Initialize weights, w and threshold, θ.
2. Present input, $x(t)$ and desired output, $d(t)$ to the perceptron.
3. Calculate actual output

$$y(t) = f_h\left(\sum_{i=0}^{n} w_i(t) x_i(t)\right)$$

4. Adapt weights

$$\begin{aligned} \Delta &= d(t) - y(t) \\ w_i(t+1) &= w_i(t) + \eta \Delta x_i(t) \\ d(t) &= \begin{cases} 1, & \text{if input from class A} \\ 0 & \text{if input from class B} \end{cases} \end{aligned}$$

Source Code

```python
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt

class Perceptron:
    def __init__(self, learning_rate=0.1, n_iters=10):
        self.lr = learning_rate
```

```python
        self.n_iters = n_iters
        self.weights = None
        self.bias = None
    def learn(self, X, y):
        n, m = X.shape
        self.weights = np.random.uniform((m, 1))
        self.bias = np.random.random_sample
        for _ in range(self.n_iters):
            f_x = np.dot(X, self.weights) + self.bias
            y_pred = self.activation_function(f_x)
            delta = self.lr * (y-y_pred)
            self.weights += delta * X
            self.bias += delta
        return
    def predict(self, X):
        linear_output = np.dot(X, self.weights) + self.bias
        y_predicted = self.activation_func(linear_output)
        return y_predicted
    def activation_function(self, x):
        return np.where(x>=0, 1, 0)

X, y = datasets.make_blobs(n_samples=1500, n_features=2, centers=2, cluster_std=1.5)
plt.scatter(X[:,0], X[:,1], c=y, marker='.')
plt.plot(X[np.argmin(X[:,0])], X[np.argmax(X[:,0])])
plt.title('Before learning')
plt.savefig('before_learning.png')
plt.show()

model = Perceptron()
model.learn(X, y)

x0_1 = np.amin(X[:, 0])
x0_2 = np.amax(X[:, 0])
x1_1 = (-model.weights[0]*x0_1 - model.bias) / model.weights[1]
x1_2 = (-model.weights[0]*x0_2 - model.bias) / model.weights[1]
ymin = np.amin(X[:, 1])
ymax = np.amax(X[:, 1])

plt.scatter(X[:,0], X[:,1], marker=".", c=y)
plt.ylim([ymin-3, ymax+3])
plt.plot([x0_1, x0_2], [x1_1, x1_2])
plt.title('After learning')
plt.savefig('after_learning.png')
plt.show()
```

Output

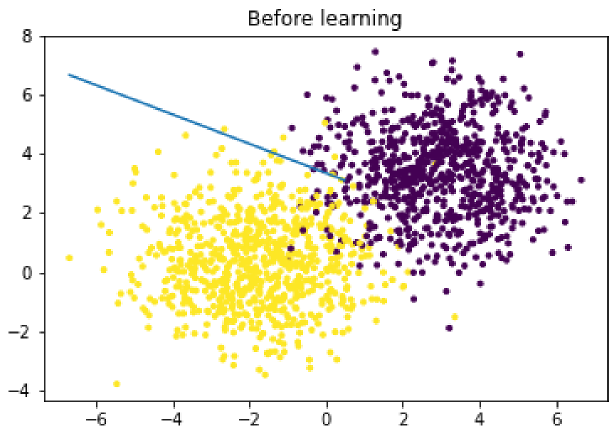

Figure 3: Before Learning Algorithm applied

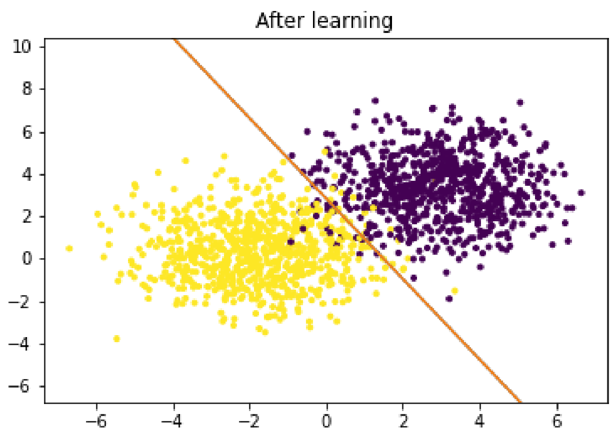

Figure 4: After Learning Algorithm applied

Advantages

1. Single Layer Perceptron is quite easy to set up and train.
2. If the data is linearly separable, then perceptron will always reach a solution in finite time.

Disadvantages

1. Can represent only a limited set of functions.
2. Only works for the linearly separable data.

Naive Bayes Algorithm

Introduction

Naive Bayes is a probabilistic machine learning algorithm based on the Bayes Theorem, used in a wide variety of classification tasks. Bayesian classification relies on the basic statistical theory of probabilities and conditional probabilities. If $G_i, i = 1, 2, \ldots, n$ be our possible list of groups, or classes, then we can define the probability of a pattern belonging to a class as $P(G_i)$, $(where\ 0 \leq P(G_i) \leq 1)$. Given a set of measurements, X, the Bayes's rule assigns a likelihood, or probability, of it belonging to a class G_i, *i.e.* $P(G_i|X)$. X belongs to class i for

$$P(G_i|X) > P(G_j|X) \quad \text{for } i = 1, 2, \ldots, n \quad i \neq j$$

i.e. we assign a pattern to the class that has the highest conditional probability of the vector X belonging to it. According to Bayes's law,

$$P(G_i|X) = \frac{P(X|G_i)P(G_i)}{\sum_j P(X|G_j)P(G_j)}$$

Source Code

```python
import numpy as np
import pandas as pd
import pprint

class NaiveBayes():
    def __init__(self, X, y):
        self.X = X
        self.y = y
        self.lookup_table = {}
    def learn(self):
        self.lookup_table['class'] = {}
        for cls in self.y.unique():
            self.lookup_table['class'][cls] = len(y[y==cls])/len(y)
        for column in self.X.columns:
            self.lookup_table[column] = {}
            for category in self.X[column].unique():
                self.lookup_table[column][category] = {}
                for cls in self.y.unique():
                    self.lookup_table[column][category][cls] =
                    ↪  len(X[y==cls][X[column]==category])/len(y[y==cls])
    def display_lookup_table(self):
        pp.pprint(self.lookup_table)
    def predict(self, x):
        prediction = {}
        p_X = 0
        for cls in self.y.unique():
            prediction[cls] = self.lookup_table['class'][cls]
            for column in self.X.columns:
```

```python
                prediction[cls] *= self.lookup_table[column][x[column]][cls]
            p_X += prediction[cls]
        for cls in prediction:
            prediction[cls] /= p_X
        return prediction

pp = pprint.PrettyPrinter()
data = pd.read_csv("https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv")
X, y = data[['Pclass', 'Sex', 'Embarked']], data['Survived']
nb = NaiveBayes(X, y)
nb.learn()
nb.display_lookup_table()
x = {'Embarked':'Q', 'Pclass':3, 'Sex':'female'}
nb.predict(x)
```

Output

```
{'Embarked': {nan: {0: 0.0, 1: 0.0},
              'C': {0: 0.1366120218579235, 1: 0.2719298245614035},
              'Q': {0: 0.08561020036429873, 1: 0.08771929824561403},
              'S': {0: 0.7777777777777778, 1: 0.6345029239766082}},
 'Pclass': {1: {0: 0.14571948998178508, 1: 0.39766081871345027},
            2: {0: 0.1766848816029144, 1: 0.2543859649122807},
            3: {0: 0.6775956284153005, 1: 0.347953216374269}},
 'Sex': {'female': {0: 0.14754098360655737, 1: 0.6812865497076024},
         'male': {0: 0.8524590163934426, 1: 0.31871345029239767}},
 'class': {0: 0.6161616161616161, 1: 0.3838383838383838}}

# calculated probabilities for the given input
{0: 0.3978469832302337, 1: 0.6021530167697664}
```

Advantages

1. It is easy and fast to predict class of test data set. It also perform well in multi class prediction.
2. It needs less training data compared to other algorithms.
3. It performs well in case of categorical features compared to numerical features.

Disadvantages

1. If categorical variable has a category in test data that was not observed in training data, then model assigns a 0 probability.