|  |
| --- |
| **Classification Algorithms** |

**Mohammed Abdul Kamran (m37 – 50290758)**

**Areebuddin Aatif Mohammed Khaja (areebudd – 50289734)**

1. **K-Nearest Neighbors Classification Algorithm**
   1. **Algorithm Flow:**

K-nearest neighbor is a lazy learning classification algorithm which classifies the data based upon the similarity between the test data record and all other records in the training data. It takes a single parameter – k – which is used to specify the number of closest neighbors that should be used to determine the label of the test record.

The algorithm flow is as follows:

* Initially the data is normalized using the z-score normalization.
* Then, dataset is split into training and evaluation data based on the k-cross validation technique.
* Then, for each record in the evaluation set, k closest neighbors are found using the distance metric.
* From these k neighbors, the majority vote is taken and the label with most votes is assigned as the label for that record.
  1. **Parametric Setting:**

As mentioned earlier, KNN takes a single parameter – k – which determines the granularity of the boundary between predicted classes. It is a hyper-parameter and is finetuned based on the training data. One thing to note is- a higher value of k will allow the classification to be more robust towards outliers but blurs the boundaries between classes and a lower value gives more crisper class boundaries but can be more prone to local maxima. Usually, for binary classification, an odd value of k is used to get a majority. In our algorithm, we set k=5 which gives the best result.

The distance metric used in our algorithm is Euclidean distance. It takes each test data record and finds its Euclidean distance to each data point in the training data and based on that determines the closest points.

* 1. **Pre-processing:**

Usually, the data is pre-processed before feeding it to the KNN algorithm to avoid certain attributes contributing more weight to the distance than others thereby skewing the results. There are several ways of normalizing the data and in our code, we’re applying z-score normalization which is done by using the mean and standard deviation of each attribute. The formula used is as follows:

**X` = (X-meann)/stdDevn**

**Where meann is the mean of the nth attribute**

**stdDevn is the standard deviation of the nth attribute**

While classification, the same mean and standard deviation can be used to normalize the test dataset.

* 1. **Point object:**

For the purpose of this project, a point class is used to store the data to help us keep track of various information related to each row of the dataset. Its definition is as follows:

class point:

def \_\_init\_\_(self, point = list(), categoricalData = list(), label=-1, groundTruth = -1, id = -1):

self.point = point

self.label = label

self.groundTruth = groundTruth

self.id = id

self.categoricalData = categoricalData

where ,

**point** is used to store all the continuous features of each row in dataset

**label** is used to store the predicted label of the data

**groundTruth** is used to store the actual label of the dataset

**categoricalData** is used to store the categorical features of the data

* 1. **Handling categorical attributes:**

Since Euclidean distance requires numeric data for calculating distance, categorical attributes need to be converted into a numerical metric. In our algorithm, we’ll be changing the categorical data into floating points using label encoding where each categorical attribute value is given a specific label.

* 1. **Implementation:**
* Initially the data is normalized using z-score normalization.

def normalizeData(self, data):

tmp = list()

for lt in data:

for point in lt:

tmp.append(point.point)

tmp = np.array(tmp)

mean = tmp.mean(axis=0)

stdDev = tmp.std(axis=0)

for lst in data:

for point in lst:

pt = list()

for i in range(len(point.point)):

pt.append((point.point[i] - mean[i])/(stdDev[i]))

point.point = np.array(pt)

return mean, stdDev

* Then for each point in the dataset its k closest neighbors are found by finding the Euclidean distance between each point in the test dataset and all the points in training dataset. Then, voting is performed based on the k closest neighbors. The majority label from these k closest neighbors is then assigned as the label for this test data.

def findLabel(self, trainData, pt, k = 3):

'''

type: trainData - a list of point objects with known labels

type: pt - a single point object whose label has to be determined

rtype: label - the predicted label for given point

'''

closestNeighbours = sorted(trainData, key = lambda x: np.linalg.norm(x.point - pt.point))[:k]

majority = defaultdict(int)

for neighbor in closestNeighbours:

majority[neighbor.groundTruth] += 1

if majority[0] > majority[1]:

return 0

else:

return 1

* 1. **Pros:**
* Evidently, this algorithm is easy to implement.
* KNN works pretty well when the data attributes are continuous.
* It is pretty fast if the number of dimensions in the data is of reasonable size
  1. **Cons:**
* The number of neighbors i.e. ‘k’ can affect the classification of the records.
* Not very useful on data with categorical and continuous attributes mixed.
* The classification is greatly affected with noisy and unrelated data.
  1. **Result Analysis:**
     1. **K-Cross Validation:**
* The validation method used here is the K-Cross Validation in which the data is split into K equal and disjoint subsets except (optionally) the last set which can consist only the remaining data.
* The process of classification is repeated K times where in each ith iteration, the ith subset of the data is used as the evaluation set and the remaining data is used for training.
* In each iteration, all the metrics i.e., Accuracy, Precision, Recall and F-Measure is calculated.
* After all K iterations, final metrics are calculated as the average of their respective measures.
* The following is the result of 10-fold cross validation:

**K (number of closest neighbors to consider) = 5**

|  |  |  |
| --- | --- | --- |
| **Measure** | **Project3\_dataset1.txt** | **Project3\_dataset2.txt** |
| **Average Accuracy** | 0.9629860601614085 | 0.6720833333333334 |
| **Average Precision** | 0.9778676470588236 | 0.562856845671491 |
| **Average Recall** | 0.9240496530725521 | 0.4371889376533339 |
| **Average F-measure** | 0.9484908488030315 | 0.47882167304987725 |

* + 1. **Analysis:**
* KNN is a lazy learner and hence requires no training.
* As it can be evidently seen from the results of dataset1, KNN works much better on continuous attributes.
* The performance is greatly affected when the dataset consists of categorical data as seen from the results of dataset2.

1. **Decision Tree classification algorithm**
2. **Naïve Bayes Classification Algorithm**
   1. **Algorithm Flow:**

Naïve Bayes is a statistical classifier which is based upon Bayes Theorem, which is stated as follows

**P(H|X) = (P(X|H) \* P(H))/P(X)**

Which states: The probability of a point being in a class is given by the product of the probability of a class consisting that point times the probability of that class from the entire dataset over the prior probability of the tuple X.

Terms,

P(H|X) is known as the Class Posterior Probability

P(X|H) is known as the Descriptor Posterior Probability

P(H) is known as the Class Prior Probability

P(X) is known as the Descriptor Prior Probability

H is the hypothesis that a tuple X belongs to class C.

This algorithm assumes that all the attributes in dataset are independent of each other. This assumption is called **class-conditional independence**.

To find the class label of each test data record, we find its class posterior probability for each class and then compare these probabilities. Out of these probabilities, whichever class label has the maximum class posterior probability is assigned to this test data record. In our algorithm implementation, we’re treating continuous and categorical attributes differently as explained in the sections below.

* 1. **Handling Continuous Attributes:**

When we come across continuous attributes, we find the mean and standard deviation for that attribute and use it to find Descriptor Posterior Probability for test data. We assume that test data follows normal distribution and calculate P(X|H) using the gaussian distribution function:

2) =

Where,

= mean of a particular attribute

= standard deviation of a particular attribute

2 = variance

* 1. **Handling Categorical Attributes:**

When we encounter categorical attribute, we find that attributes probability by counting the occurrences of each value in that attribute for that particular class and dividing it by the total number of records.

* 1. **Handling Zero Probability:**

There can be instances where the test dataset has an attribute value which is not present in the training dataset which can result in getting a zero descriptor posterior probability. To handle such instances, we will be using Gaussian Distribution Function to determine descriptor posterior probabilities.

* 1. **Implementation:**
* First, we find the class priori probabilities as follows:

def findClassPriorProbability(self, data):

'''

:type data- a list of Point objects

:rtype res- a dictionary with key as class label and value as its probability

'''

class\_map = defaultdict(int)

for pt in data:

class\_map[pt.groundTruth] += 1

res = dict()

for key in class\_map:

res[key] = class\_map[key]/len(data)

return res

* Then, we find the mean and standard deviation for each numerical attribute and probability of occurrences for categorical attributes.

def findDescriptorPosteriorProbabilites(self, classes, td):

occurences = defaultdict(int)

mean, stdDeviation = defaultdict(dict), defaultdict(dict)

for key in classes:

tmp = classes[key]

mean[key], stdDeviation[key] = hp().standardizeBayes(tmp)

for pt in tmp:

for index, i in enumerate(pt.categoricalData):

if (i, key) not in occurences:

count = self.countOccurence(i, index, tmp)

occurences[(i, key)] = count/len(tmp)

return occurences, mean, stdDeviation

* Finally, for each point in the test data, we determine its label by choosing the label with maximum probability.

def bayesProbabilty(self, point, ph, occurences, mean, stdDeviation):

maxProbability = float('-inf')

label = -1

for key in ph:

phi = ph[key]

probability = 1.0

for index, i in enumerate(point.point):

den = 2\*(22/7)\*(stdDeviation[key][index]\*\*2)

num = ((i-mean[key][index])\*\*2) / (2\*(stdDeviation[key][index]\*\*2))

probability\*=(1/den\*\*0.5)\*(e\*\*(-1\*num))

for index, i in enumerate(point.categoricalData):

probability\*=occurences[(i, key)]

probability\*=phi

if probability >= maxProbability:

maxProbability = probability

label = key

return label

* 1. **Pros:**
* If the conditional independence assumption holds true, then this classifier can be really efficient and can give really impressive results.
* It can handle continuous attributes as well as categorical attributes.
* It makes probabilistic predictions.
  1. **Cons:**
* If the conditional independence assumption does not hold, then the algorithm does not work well.
* It faces a ‘zero probability’ issue if there is a certain value in the test data that is not present in the training data. However, this issue can be handled using Laplacian Correction.
  1. **Result Analysis:**
     1. **K-Cross Validation:**

The following is the result of 10-fold cross validation:

|  |  |  |
| --- | --- | --- |
| **Measure** | **Project3\_dataset1.txt** | **Project3\_dataset2.txt** |
| **Average Accuracy** | 0.9357142857142857 | 0.7043478260869564 |
| **Average Precision** | 0.9241315440217043 | 0.57573192631242165 |
| **Average Recall** | 0.9053435453435455 | 0.617493091703618 |
| **Average F-Score** | 0.913209052226269 | 0.58571298120524126 |

* + 1. **Analysis:**
* As it can be seen, this algorithm performs well for continuous data as well as categorical data.
* The algorithm performs fairly efficiently even for large datasets.
* Naïve Bayes performs relatively better for dataset2 which consists of both numerical and categorical attributes.