**CSE 601**

**DATA MINING AND BIO INFORMATICS**

**PROJECT 3 – REPORT**

**IMPLEMENTATION OF Classifiers**

**SUBMITTED BY:**

**ARTI GUPTA – 50170010**

**artigupt@buffalo.edu**

**SUDHARCHITH SONTY - 50169912**

[**sudharch@buffalo.edu**](mailto:sudharch@buffalo.edu)

**K-NEAREST NEIGHBOURS CLUSTERING ALGORITHM**

**UNDERSTANDING**

The ***k*-Nearest Neighbors algorithm** (or ***k*-NN** for short) is a [non-parametric](https://en.wikipedia.org/wiki/Non-parametric_statistics) method used for [classification](https://en.wikipedia.org/wiki/Statistical_classification) and [regression](https://en.wikipedia.org/wiki/Regression_analysis).[[1]](https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm#cite_note-1) In both cases, the input consists of the *k* closest training examples in the [feature space](https://en.wikipedia.org/wiki/Feature_space). The output depends on whether *k*-NN is used for classification or regression:

* In *k-NN classification*, the output is a class membership. An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its *k* nearest neighbors (*k* is a positive [integer](https://en.wikipedia.org/wiki/Integer), typically small). If *k* = 1, then the object is simply assigned to the class of that single nearest neighbor.
* In *k-NN regression*, the output is the property value for the object. This value is the average of the values of its *k* nearest neighbors.

*k*-NN is a type of [instance-based learning](https://en.wikipedia.org/wiki/Instance-based_learning), or [lazy learning](https://en.wikipedia.org/wiki/Lazy_learning), where the function is only approximated locally and all computation is deferred until classification. The *k*-NN algorithm is among the simplest of all [machine learning](https://en.wikipedia.org/wiki/Machine_learning) algorithms.

Both for classification and regression, it can be useful to assign weight to the contributions of the neighbors, so that the nearer neighbors contribute more to the average than the more distant ones.

The neighbors are taken from a set of objects for which the class (for *k*-NN classification) or the object property value (for *k*-NN regression) is known. This can be thought of as the training set for the algorithm, though no explicit training step is required.

**Implementation Details:**

* We have used JAVA for the implementation of KNN Algorithm.
* The following methods have been used and called as a part of the implementation.

readFile (String input)

*Parse the input data from .txt file and store into ArrayList of objects which map to a class with two attributes the dimensions and the classname.*

splitdata(input)

*This function is used to perform the 10-fold split for cross validation.*

*The split is performed by dividing the data into 10 parts, and then selecting each part randomly.*

Find majority element()

*This method is used for finding out the majority class by consensus. Ties are broken by choosing an element at random.*

calculate\_distance()

*function to get Euclidean Distance*

calculatestatistics(list)

*Function to get the average of the cumulative performance measures.*

**CLASSES:**

*Class points: attributes are list of dimesions and classname*

*Class result: attributes are distance and classname.*

**STEPS:**

1. The algorithm first calls the parse function, to generate a list of objects which are instances of the points class described above from the input file.
2. Then, we call the split method to split the data as per our k-fold validation requirement.
3. We, take a random variable and assign the corresponding set of values i.e.; list(get) to test and the other 9 parts as training.
4. While, we are looping through the data, we make sure that the condition is that the test data is not equal to the training data to avoid confusion
5. For each, n-dimensional point in the test, we compute the Euclidean distance with respect to each point in the training and put the distance along with its corresponding class in the result list
6. Then, we get sort the distance list and get the class names of the top k distances and transfer them into an array.
7. Next, we call the Majority element method and get the top repeated terms from the array and ties are broken by choosing an element at random.
8. The point is assigned its new class and then we check for equality between the new class and the old class.
9. Based on the values we update the TP, TN, FP, FN values and get the values
10. At the end of the final run, (number of folds) we calculate the cumulative accuracy and other performance measures
11. The algorithm automatically stops, when the number of iterations reaches the maximum number of iterations.

**Datasets Used and Results Obtained:**

1. **Project3\_dataset1.txt –** 
   1. *Run1, k =5*

---KNN---

Accuracy: 0.6099479049285792

Precision: 0.762852869698203

Recall: 0.7542225490958683

F-Measure: 0.757581022134739

* 1. *Run2, k = 6*

---KNN---

Accuracy: 0.6188477783475991

Precision: 0.7748025534633655

Recall: 0.7403323468284013

F-Measure: 0.756637193836865

* 1. *Run3, k =4*

---KNN---

Accuracy: 0.8633178322563746

Precision: 1.0

Recall: 0.5680530699418529

F-Measure: 0.7236167666260679

1. **Project3\_dataset2.txt –**
   1. *Run1, k =7*

Accuracy: 0.8024542080587546

Precision: 1.0

Recall: 0.3820326878967746

F-Measure: 0.552421651974945

* 1. Run2, k = 4

Accuracy: 0.8437780970263649

Precision: 1.0

Recall: 0.5404279765480882

F-Measure: 0.6977405006146332

* 1. *Run3, k =9*

C Accuracy: 0.7971780632900626

Precision: 1.0

Recall: 0.3704287730078938

F-Measure: 0.5390405607582063

**PROS:**

* Simple to implement
* Flexible to feature / distance choices
* Naturally handles multi-class cases
* Can do well in practice with enough representative data

**CONS:**

* Large search problem to find nearest neighbours
* Storage of data
* Must know we have a meaningful distance function
* it is sensitive to the local structure of the data

**DECISION TREE ALGORITHM:**

**UNDERSTANDING:**

* A decision tree is a flowchart-like tree structure, where each internal node (nonleaf node) denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (or terminal node) holds a class label. The topmost node in a tree is the root node
* Internal nodes are denoted by rectangles, and leaf nodes are denoted by ovals. Some decision tree algorithms produce only binary trees (where each internal node branches to exactly two other nodes), whereas others can produce nonbinary trees.
* The algorithm uses the same process recursively to form a decision tree for the tuples at each resulting partition, Dj , of D .
* The recursive partitioning stops only when any one of the following terminating conditions is true:
* All the tuples in partition D (represented at node N) belong to the same class (steps 2 and 3).
* There are no remaining attributes on which the tuples may be further partitioned (step 4). In this case, majority voting is employed (step 5). This involves converting node N into a leaf and labelling it with the most common class in D. Alternatively, the class distribution of the node tuples may be stored. 3.
* There are no tuples for a given branch, that is, a partition Dj is empty (step 12). In this case, a leaf is created with the majority class in D (step 13). The resulting decision tree is returned .

**Implementation Details:**

* Decision Tree was implemented using the Hunts Algorithm and the Algorithm is mentioned below.
* The methods used are:

**STEPS:**

1. The algorithm first calls the read\_data function to read the input file data
2. Then, the algorithm computes the distance matrix using numpy library.
3. It makes clusters, starting with the pair of points which have minimum distance between them; (plist).
4. Then, it keeps merging clusters into altogether till the algo is asked to stop after certain points; (n-5) cluster etc.
5. Finally, we get one cluster comprises of several small clusters.

**Datasets Used and Results Obtained:**

1. **Project3\_dataset1.txt –**

---10 FOLD CROSS VALIDATION DECISION TREE---

accuracy: 0.8928571428571429

Precision: 0.9375

Recall: 1.0

F-Measure: 0.8333333333333334

---10 FOLD CROSS VALIDATION RANDOM FOREST---

Accuracy: 0.39215686274509803

Precision: 0.39215686274509803

Recall: 1.0

F-Measure: 0.5633802816901409

1. **Project3\_dataset2.txt -**

---10 FOLD CROSS VALIDATION DECISION TREE---

accuracy: 0.7555555555555555

Precision: 0.6956521739130435

Recall: 0.7619047619047619

F-Measure: 0.7272727272727272

---10 FOLD CROSS VALIDATION RANDOM FOREST---

Accuracy: 0.5853658536585366

Precision: 0.625

Recall: 0.2631578947368421

F-Measure: 0.37037037037037035

**PROS:**

* It can produce an ordering of the objects, which may be informative for data display.
* Smaller clusters are generated, which may be helpful for discovery.
* It shows more quality as opposed to K-means.
* Does not need number of clusters to be pre-specified.
* With hierarchical clustering, you will most definitely get the same clustering results.
* Hierarchical Clustering can give a diﬀerent partitioning of data depending on the level-of-resolution.

**CONS:**

* Works well only when the data set is relatively small.
* For, large datasets, the performance decreases and the execution time increases. O(n2).
* No provision can be made for a relocation of objects that may have been 'incorrectly' grouped at an early stage.
* Use of different distance metrics for measuring distances between clusters may generate different results.

**NAÏVE BAYES ALGORITHM:**

**Implementation Details:**

**STEPS:**

1. The algorithm first read the input text file.
2. Then, it calculates its neighbour.
3. It calculates DBSCAN algorithm, where, it adds each point to visited if it has been visited in data.
4. Else, it adds that point to Noise and expand algorithm further.
5. In expandcluster function, it again proceed with points which are not visited, then add them and continue processing like that.
6. Finally, it calculates groundtruth values.
7. Jaccard score and random index has been calculated.

In k-fold cross-validation, the initial data are randomly partitioned into k mutually exclusive subsets or “folds,” D1, D2,..., Dk , each of approximately equal size. Training and testing is performed k times. In iteration i, partition Di is reserved as the test set, and the remaining partitions are collectively used to train the model. That is, in the first iteration, subsets D2,..., Dk collectively serve as the training set to obtain a first model, which is tested on D1; the second iteration is trained on subsets D1, D3,..., Dk and tested on D2; and so on. Unlike the holdout and random subsampling methods, here each sample is used the same number of times for training and once for testing. For classification, the accuracy estimate is the overall number of correct classifications from the k iterations, divided by the total number of tuples in the initial data.

Boosting. In boosting, weights are also assigned to each training tuple. A series of k classifiers is iteratively learned. After a classifier, Mi , is learned, the weights are updated to allow the subsequent classifier, Mi+1, to “pay more attention” to the training tuples that were misclassified by Mi . The final boosted classifier, M∗, combines the votes of each individual classifier, where the weight of each classifier’s vote is a function of its accuracy. AdaBoost (short for Adaptive Boosting) is a popular boosting algorithm. Suppose we want to boost the accuracy of a learning method. We are given D, a data set of d class-labeled tuples, (X1, y1),(X2, y2),...,(Xd, yd), where yi is the class label of tuple Xi . Initially, AdaBoost assigns each training tuple an equal weight of 1/d. Generating k classifiers for the ensemble requires k rounds through the rest of the algorithm. In round i, the tuples from D are sampled to form a training set, Di , of size d. Sampling HAN 15-ch08-327-392-9780123814791 2011/6/1 3:21 Page 381 #55 8.6 Techniques to Improve Classification Accuracy 381 with replacement is used—the same tuple may be selected more than once. Each tuple’s chance of being selected is based on its weight. A classifier model, Mi , is derived from the training tuples of Di . Its error is then calculated using Di as a test set. The weights of the training tuples are then adjusted according to how they were classified. If a tuple was incorrectly classified, its weight is increased. If a tuple was correctly classified, its weight is decreased. A tuple’s weight reflects how difficult it is to classify— the higher the weight, the more often it has been misclassified. These weights will be used to generate the training samples for the classifier of the next round. The basic idea is that when we build a classifier, we want it to focus more on the misclassified tuples of the previous round. Some classifiers may be better at classifying some “difficult” tuples than others. In this way, we build a series of classifiers that complement each other.

We now present another ensemble method called random forests. Imagine that each of the classifiers in the ensemble is a decision tree classifier so that the collection of classifiers HAN 15-ch08-327-392-9780123814791 2011/6/1 3:21 Page 383 #57 8.6 Techniques to Improve Classification Accuracy 383 is a “forest.” The individual decision trees are generated using a random selection of attributes at each node to determine the split. More formally, each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest. During classification, each tree votes and the most popular class is returned. Random forests can be built using bagging (Section 8.6.2) in tandem with random attribute selection. A training set, D, of d tuples is given. The general procedure to generate k decision trees for the ensemble is as follows. For each iteration, i(i = 1, 2,..., k), a training set, Di , of d tuples is sampled with replacement from D. That is, each Di is a bootstrap sample of D (Section 8.5.4), so that some tuples may occur more than once in Di , while others may be excluded. Let F be the number of attributes to be used to determine the split at each node, where F is much smaller than the number of available attributes. To construct a decision tree classifier, Mi , randomly select, at each node, F attributes as candidates for the split at the node. The CART methodology is used to grow the trees. The trees are grown to maximum size and are not pruned. Random forests formed this way, with random input selection, are called Forest-RI. Another form of random forest, called Forest-RC, uses random linear combinations of the input attributes. Instead of randomly selecting a subset of the attributes, it creates new attributes (or features) that are a linear combination of the existing attributes. That is, an attribute is generated by specifying L, the number of original attributes to be combined. At a given node, L attributes are randomly selected and added together with coefficients that are uniform random numbers on [−1,1]. F linear combinations are generated, and a search is made over these for the best split. This form of random forest is useful when there are only a few attributes available, so as to reduce the correlation between individual classifiers.

**Datasets Used and Results Obtained:**

1. **Cho.txt –**

DBSCAN(4,80)

eps = 4

min\_pts = 80

jaccard\_coefficient is 0.176165803109

adjusted\_rand\_index is 0.32136703885094814

DBSCAN(3,3)

eps = 3

min\_pts = 3

jaccard\_coefficient is 0.383419689119

adjusted\_rand\_index is 0.47340450713557203

DBSCAN(1.8,20)

eps = 1.8

min\_pts = 20

jaccard\_coefficient is 0.142487046632

adjusted\_rand\_index is 0.16053228077880788

1. **Iyer.txt –**

DBSCAN(4,80)

eps = 4

min\_pts = 80

jaccard\_coefficient is 0.243713733075

adjusted\_rand\_index is 0.2321465145855259

DBSCAN(3, 3)

eps = 4

min\_pts = 80

jaccard\_coefficient is 0.193423597679

adjusted\_rand\_index is 0.3168913262717644

DBSCAN(1.8,20)

eps = 1.8

min\_pts = 20

jaccard\_coefficient is 0.2166344294

adjusted\_rand\_index is 0.390976042124015

**PROS:**

* DB Scan algorithms produces clusters of arbitrary shape.
* The algorithm, to a large degree is robust to noise.
* The number of clusters does not have to be pre specified.
* DB Scan requires just two parameters and is mostly insensitive to the ordering of the points in the database.

**CONS:**

* Deterministic Data sets with varying densities are problematic.
* Requires connected regions of sufficiently high density.
* DB Scan cannot cluster data sets well with large differences in densities.

**COMPARISION BETWEEN DIFFERENT ALGORITHMS:**

|  |  |  |  |
| --- | --- | --- | --- |
| ATTRIBUTE | K-NN | DECISION  TREE | NAÏVE  BAYES |
| RUNNING TIME | O(kn), k = number of clusters, n is number of points | O (n × |D| × log(|D|)) | O(n), n is the number of points |
| SIZE OF DATA SET | Works well on Large and small datasets | Works well on small data sets | Works well on smaller data sets |
| VALUE OF K(number of cluster) | Pre-Specified | Dynamic | Dynamic |
| ACCURACY | Hyper-spherical | Arbitrary | Arbitrary |
| RUNTIME SCALING | Maintains a linear runtime. Does scale up in large data sets but still is better than Hierarchical. | Scales up to quadratic runtime. | Worst case, it can scale up to quadratic runtime. |
| CONSISTENCY | Random centroids, leads to random results for each run | More or less the same clusters for each iteration, thus being consistent | Is insensitive to ordering and produces different results based on parameters. |
| PRECISION | Outliers have some effect | Noisy data has effect on result. | Robust towards Noise |
| RECALL | For new\_dataset\_1  JC = 0.56 | For new\_dataset\_2  JC = | For new\_dataset\_1  JC = |

**RANDOM FORESTS:**

**Implementation Details:**

**PREPROCESSING STAGE:**

* In the preprocessing stage of the Algorithm, we use two functions,

**parseInputData :** to read from the file and create an input data map with key = geneId and value = list(expression values)

Running Time: O(n)

**getrandomcentroids:** to read from the inputdata map and select 5 random values and create initial centroids that will be used for the first iteration of the Map Reduce K-Means Algorithm

Running Time: O(k), k = initial value

Number of Mappers is 1

Number of Reducers is 1

**MAPPER FUNCTION:**

* The Mapper function, takes as input each line from the input file(new\_dataset\_1) and extracts the geneId(point) and the list of expression values (expressionvalues)
* For the extracted expression values, we calculate the Euclidean distance of each n-Dimensional point with the initial centroids, computed by the Calculate Distance Method.
* The computed distance values are then compared with each other and for each point, the centroid closest to it is marked as it Cluster ID.
* We then write, the closestcluster(Cluster ID) and the point corresponding to it and send it to the Reducer.
* Running Time of Mapper Phase: O(n), n is the length of input file.

**REDUCER FUNCTION:**

* The input to the reducer function is a tuple which has the point (Cluster ID) and a list of values (points belonging to that cluster).
* We then iterate over the set of points and for each point, we get the list of values associated with that point.
* We then compute the mean of all the expression values thus obtained and store the mean in a temporary list.
* We then update the final\_initial\_centroids map created in the Pre-Processing Stage to reflect the updated centroids.
* The updated map is then fed as the input to the Mapper again to recomputed the clusters.

1. The program runs till the maximum iterations have been reached.
2. If, during the iterations, for a given iteration, the set of previous centroids is equal to the set of new centroids, then the program stops and outputs that as the result.
3. Total Running Time is O(cn), where c is the number of iterations that are needed.

**Datasets Used and Results Obtained:**

**Dataset Used: iyer.txt**

**Result Obtained:**

**JACCARD COEFFICIENT IS: 0.27**

**Dataset Used: cho.txt**

**Result Obtained:**

**JACCARD COEFFICIENT IS: 0.25**

**IMPROVEMENTS THAT CAN BE MADE and OBSERVATIONS:**

* Improvements in the performance of the K means algorithm, is to use multiple mappers and reducers as opposed to the Single mapper and Single Reducer that we have implemented.
* **SPEEDUP:** To measure the speedup, we can keep the dataset constant and increase the number of computers in the system. The perfect parallel algorithm demonstrates linear speedup: a system with m times the number of computers yields a speedup of m. However, linear speedup is difficult to achieve because the communication cost increases with the number of clusters becomes large.
* **SIZEUP:** The Size up factor can be measured, if we keep the Number of computers constant and keep increasing the size of the dataset. Since, K means generally works well on larger data sets, Parallel K means also works increasingly well
* **SCALABILITY:** Scalability factor can be measured by increasing the size of the datasets, proportional to the number of the computers.