CSE 601 Data Mining and Bioinformatics

Project 3: Classification Algorithms

Submitted By
Mayur Tale
UB IT Name: mayurvin
Person # 50169256

Nearest Neighbor

<u>Introduction:</u>

The *k*---nearest neighbor's algorithm (*k*---NN) is a non---parametric method for classification. That predicts objects' "values" or class memberships based on the *k* closest training examples in the feature space. The *k*—nearest neighbor algorithm is amongst the simplest of all machine learning algorithms: an object is classified by a majority vote of its neighbors, with the object being assigned to the class most common amongst its *k* nearest neighbors. If *k* = 1, then the object is simply assigned to the class of that single nearest neighbor.

<u>Algorithm:</u>

The main data set is read from txt file and divided into training and testing data set. The training and testing examples are vectors in a multidimensional feature space, each with a class label. In the classification phase, an unlabeled vector is classified by assigning the label which is most frequent among the *k* training samples nearest to that query point. A commonly used distance metric, that I used for this algorithm, is Euclidean distance. To classify an unknown record:

1. Compute distance to other training records
2. Identify *k* nearest neighbors
3. determine the class label of unknown record by taking majority vote
4. let the closest points among the *K* nearest neighbors have more say in affecting the outcome of the query point. This can be achieved by introducing a set of weights *W*, one for each nearest neighbor, defined by the relative closeness of each neighbor with respect to the query point. Thus determine the class labels according to Weight factor, w = 1/d^2

<u>Flow of Code:</u>

readFiles () - Reads the txt files and puts the data into the ArrayList<ArrayList<Integer>> as dataset1 and dataset2. For string value "PRESENT" it adds 1, and for "ABSENT" it adds 0 of type double.
Main () – Divides the data into Training and Test part. I am taking 30% of data as test data and 70% as training data at first. It then processes the test and train data and displays the result. Then it Calls the getAllParameters () method.
getAllParameters (arguments) - This is the main function to start calculating the accuracy, precision, recall and F-measure parameters. It is called once for each dataset.
NormalizeDatasets (arguments) - It normalizes the data by using MAX and MIN double values for training and testing data.
A list of nearest neighbors is then created using the Euclidean distance between points which is calculated by getDistance () method provided the training dataset.
getEvaluationParameters () – Given the ArrayList of nearest neighbors and their distance list, calculates the count of zeros and ones according to weights and then assigns the label greater of the two (Zero or One). This label is then used to calculate the accuracy, precision, recall and F-measure parameters which are returned as a list of double variables.

CrossValidate () ---
It performs the 10-fold cross validation on the data, by setting one part as test data, and rest as train data at each time. The data is divided in random parts, by adding the leftover data buffer into each partition.

Parameters:

The best choice of *k* depends upon the data; generally, larger values of *k* reduce the effect of noise on the classification, but make boundaries between classes less distinct.

Estimation through Cross validation:

The general idea is to divide the data sample into a number of *10* folds randomly. For a fixed value of *k*, we apply the *KNN* model to make predictions on the *i*th segment and evaluate the error as the accuracy. This process is then successively applied to all possible choices of i. At the end of the 10 folds, the computed errors are averaged to yield a measure of the stability of the model. The above steps are then repeated for various *k* and the value achieving the highest classification accuracy is then selected as the optimal value or *k* which in our case comes out to be **k=5**.

Pros:

1. Simplest of all to evaluate.
2. Do not need training of data.

Cons:

1. Accuracy of algorithm can be degraded by presence of noisy or irrelevant features.
2. It does not build models explicitly.
3. Different from eager learners such as decision tree induction.
4. Classifying unknown records are relatively expensive.

Results:

Nearest Neighbor Performance Parameters

Parameter	Dataset 1	Dataset 2
Accuracy	0.9647058823529412	0.7246376811594203
Precision	0.9272727272727272	0.6666666666666666
Recall	0.9622641509433962	0.5384615384615384
F-Measure	0.9444444444444444	0.5957446808510638

10 Fold Cross Validation Average Values

Parameter	Dataset 1	Dataset 2
Accuracy	0.9509839315334206	0.6583105547631458
Precision	0.9537034523805341	0.5370620872215766
Recall	0.9159889561386138	0.4003222276036714
F-Measure	0.9328809731904182	0.4470696913230131

Decision Tree

Introduction:

Decision Tree represents M-ary node tree of decision or features and their values along with the corresponding resultant class. It is a graphical form of multiple and/or nested if else statements with the conditions being the attribute values. It is useful for classifying both numeric and categorical data. Training data is used to build the tree by selecting the most useful (maximum variance) features at each step. Different measures are used to select these features. The testing step involve traversing the tree along matching attribute values until leaf node is reached. The leaf node contains the actual result of classification.

Algorithm:

Decision tree implemented here uses multi-way split to reduce number of branches and hence computation and storage. Entropy based information gain was used to split data. The entire data set is divided into train and test set in 70:30 ratios. The data is split randomly splits the data input row as train or test using a random number generator. It returns the corresponding datasets and class Labels.

The train data and labels are fed to DecisionTree class and create tree method is called. DecisionTreeNode represents a node of tree containing list of children, remaining attributes decision if any and rows in that sub tree. A root node is first created and prior entropy is calculated for entire train set (before split). Information gain is then computed for each attribute and one giving highest gain is chosen for split. Next this is repeated for each distinct values of chosen attribute and split performed on one giving highest gain from remaining attributes. The process is continued till a leaf node is reached. To avoid stack overflow in case of a very large set level order traversal similar to BFS (breadth first search) is followed instead of recursion.

A leaf node either has all values falling in the same class or has no more attributes remaining. If leaf node is such that no unused attributes remaining and all inputs do not fall in one class, then the one which has the maximum probability is assigned as the decision for that class. Threshold can be specified for information gain. If gain is below threshold that node is treated as leaf. Pruning in this way avoids unnecessary computations. In testing the tree is traversed from root towards leaf along corresponding attribute values and class is assigned. This can be done using the traverse Tree method which returns the decision.

Flow of Code:

readFiles () - Reads the txt files and puts the data into the ArrayList<ArrayList<Integer>> as dataset1 and dataset2. For string value "PRESENT" it adds 1, and for "ABSENT" it adds 0 of type double.

Main () – Divides the data into Training and Test part. I am taking 30% of data as test data and 70% as training data at first. It then processes the test and train data and creates the data matrices for each training and testing datasets. Then it Calls the getAllParameters () method.

getAllParameters (arguments) - This is the main function to start building the binary tree and calculating the accuracy, precision, recall and F-measure parameters. It is called once for each dataset.

createBinaryTree (arguments) – this method calls the createBinaryNode () method which first creates a root node for the binary tree and then recursively creates Binary tree nodes and their children and add these nodes to the root of the main binary tree.

createListOfGiniObj () – An ArrayList of Gini values is created for each column which is used to get the column with minimum gini value which is then used to split the data for left and right child nodes. We used Gini as the measure of splitting the continuous data given in the dataset. The split position is determined on each node as, by sorting the data based on, each, individual column to determine its min Gini split value by finding all gini value between each change in dataset value. And then compare all the values to find the column which has the minimum gini split value, this column is then used as the splitting attribute.

CrossValidate ()

It performs the 10-fold cross validation on the data, by setting one part as test data, and rest as train data at each time. The data is divided in random parts, by adding the leftover data buffer into each partition. Then average values for each fold of rotation are calculated for accuracy, precision, recall and F-measure.

Pros:

a. Works best for exploratory knowledge discovery and mining information
b. Can handle multidimensional data
c. After training, classifications can be simplified to restructure tree
d. Visualization in tree form and classification rules are easy to understand
e. Training accuracy is high when tree is deep/large

Cons:

a. Doesn't work for inputs that haven't been seen
b. Doesn't suit for continuous data – have to discretize data
c. Doesn't work well too many classes.

Results:

Decision Tree Performance Parameters

Parameter	Dataset 1	Dataset 2
Accuracy	0.9294117647058824	0.6406926406926406
Precision	0.8727272727272727	0.5362318840579711
Recall	0.9056603773584906	0.42045454545454547
F-Measure	0.8888888888888888	0.4713375796178344

10 Fold Cross Validation Average Values

Parameter	Dataset 1	Dataset 2
Accuracy	0.928564399405278	0.6631957050610526
Precision	0.8744138472005675	0.37296298286775376
Recall	0.9412613784530173	0.37398164327064654
F-Measure	0.9050791177776235	0.3676300805232131

Naïve Bayes Classifier

Introduction:

A naïve Bayes classifier is a simple probabilistic classifier based on applying Bayes' theorem with strong (naive) independence assumptions. It assumes that the presence or absence of a particular feature is unrelated to the presence or absence of any other feature, given the class variable. Parameter estimation for naïve Bayes models uses the method of maximum likelihood. Naïve Bayes classifiers can handle an arbitrary number of independent variables whether continuous or categorical.

Algorithm:

In Naïve Bayes algorithm, we feed the system with a set of train data and expect it to predict the result for new data. If we pass data points from the training set for testing the algorithm, the algorithm may give a different result than what it was trained against. The probability of data point correctly classified into the right category depends on the prior and the posterior probability.

Assuming independence of attributes, we calculate the occurrence of each attribute value corresponding to each class i.e. frequency based probabilities are obtained. This is also stored in map which records each value of each of the features in the training set and their probabilities. Each test input is used to calculate posterior probability of data belonging to a particular class. The class whose posterior is maximum is chosen as the class label. To calculate posterior each of the features' value is read from input tuple and corresponding probability is obtained from the map. Multiplying features we get the likelihood of input which is multiplied prior to get posterior.

With the current set of variables V = {v1, v2, v3..., v30}, we compute the posterior probability for the event Cj among a set of C = {0, 1}

$$P\ (Cj|\ v1, v2, v3, \ldots, v30) = p\ (v1, v2, v3, \ldots, v30|Cj)$$

However, since we assume, during calculation of prior, that all the attribute values are independent of each other.

$$P\ (X|Cj) = \Pi\ p\ (xk|Cj)$$

or

$$p\ (Cj|X) = p\ (Cj)\ \Pi p\ (xk|Cj)$$

Flow of Code:

readFiles () - Reads the txt files and puts the data into the ArrayList<ArrayList<Integer>> as dataset1 and dataset2. For string value "PRESENT" it adds 1, and for "ABSENT" it adds 0 of type double.

Main () – Divides the data into Training and Test part. I am taking 30% of data as test data and 70% as training data at first. It then processes the test and train data and displays the result.

randomizeAndMatrixize **(arguments) -** This function is called once for each dataset. It creates the class "Zero" and class "One" label matrices and return them to the main function for further processing along with training and testing data matrices of type double.

generateMetaData (arguments) – This function takes the class Zero and class One matrices as input and creates and returns ArrayList of HashMap of ArrayList of mean and standard deviation.

Then for each row in test data matrix Gaussian probability is calculated using the guassianProb () method depending on whether the label is Zero or One. Then a list of predicted labels is created using the predicted label as the one that has greater guassian probability.

Given the ArrayList of predicted labels for each row in the dataset, accuracy, precision, recall and F-measure are calculated using the true labels in the test data matrix.

CrossValidate ()
It performs the 10-fold cross validation on the data, by setting one part as test data, and rest as train data at each time. The data is divided in random parts, by adding the leftover data buffer into each partition. It repeats the above procedure and calculates the average over ten-folds and prints it to the screen for each dataset.

Zero probability:

The Laplacian correction (or Laplace estimator) is a way of dealing with zero probability values. problem. I assumed that our training set is so large that adding one to each count that we need would only make a negligible difference in the estimated probabilities, yet would avoid the case of zero probability values.

Pros:

1. Fast learning – one data seen => one data learned.
2. Works well for discrete and continuous data.
3. Can classify datasets for training datasets as small as 1!!
4. Works well for unseen data too (accuracy is probabilistically determined).
5. Easy to design and implement

Cons:

1. Assumption that all the attributes are independent!
2. Need to find correlation between attributes, if any.
3. Doesn't work for new labels
4. Repeated training data may bias the result towards one label.

Results:

10 Fold Cross Validation Average Values for Naïve Bayes Classifier

Parameter	Dataset 1	Dataset 2
Accuracy	0.931764705882353	0.6804347826086957
Precision	0.3767469900082231	0.6527099605168694
Recall	0.389196960249091	0.5324132210807421
F-Measure	0.38222763820422706	0.5849753065873969

Random Forest

Introduction:

The random forest starts with a “decision tree” which, in ensemble terms, corresponds to our weak learner. In a decision tree, an input is entered at the top and as it traverses down the tree the data gets bucketed into smaller and smaller sets. The random forest takes this notion to the next level by combining trees with the notion of an ensemble. Thus, in ensemble terms, the trees are weak learners and the random forest is a strong learner.
The train data and labels are fed to DecisionTree class and create tree method is called. DecisionTreeNode represents a node of tree containing list of children, remaining attributes decision if any and rows in that sub tree. A root node is first created and prior entropy is calculated for entire train set (before split). Information gain is then computed for each attribute and one giving highest gain is chosen for split. Next this is repeated for each distinct values of chosen attribute and split performed on one giving highest gain from remaining attributes. The process is continued till a leaf node is reached. To avoid stack overflow in case of a very large set level order traversal similar to BFS (breadth first search) is followed instead of recursion.

Algorithm:

Step 1. Sample *N* cases at random with replacement to create a subset of the data. The subset should be about 66% of the total set.
Step 2. At each node:

1. For some number *m* (see below), *m* predictor variables are selected at random from all the predictor variables.
2. The predictor variable that provides the best split, according to some objective function, is used to do a binary split on that node.
3. At the next node, choose other m variables at random from all predictor variables and do the same.

Flow of Code:

readFiles () - Reads the txt files and puts the data into the ArrayList<ArrayList<Integer>> as dataset1 and dataset2. For string value “PRESENT” it adds 1, and for “ABSENT” it adds 0 of type double.
Main () – Divides the data into Training and Test part. I am taking 30% of data as test data and 70% as training data at first. It then processes the test and train data and creates the data matrices for each traning and testing datasets. Then it Calls the getAllParameters () method.
getAllParameters (arguments) - This is the main function to start building the binary tree and calculating the accuracy, precision, recall and F-measure parameters. It is called once for each dataset.
createBinaryTree (arguments) – this method calls the createBinaryNode () method which first creates a root node for the binary tree and then recursively creates Binary tree nodes and their children and add these nodes to the root of the main binary tree.

createListOfGiniObj () – An ArrayList of Gini values is created for each column which is used to get the column with minimum gini value which is then used to split the data for left and right child nodes. We used Gini as the measure of splitting the continuous data given in the dataset. The split position is determined on each node as, by sorting the data based on, each, individual column to determine its min Gini split value by finding all gini value between each change in dataset value. And then compare all the values to find the column which has the minimum gini split value, this column is then used as the splitting attribute.

CrossValidate ()

It performs the 10-fold cross validation on the data, by setting one part as test data, and rest as train data at each time. The data is divided in random parts, by adding the leftover data buffer into each partition. Then average values for each fold of rotation are calculated for accuracy, precision, recall and F-measure.

Pros:

1. It runs efficiently on large data bases
2. It gives estimates of what variables are important in the classification
3. It is unexcelled in accuracy among current algorithms

Cons:

1. Doesn't work for inputs that haven't been seen
2. Doesn't suit for continuous data – have to discretize data
3. Doesn't work well too many classes.

Results:

10 Fold Cross Validation Average Values for Random Forest Algorithm

Parameter	Dataset 1	Dataset 2
Accuracy	0.9507042253521126	0.6260869565217392
Precision	0.9361702127659575	0.5166666666666667
Recall	0.9166666666666666	0.6888888888888889
F-Measure	0.9263157894736843	0.5904761904761905

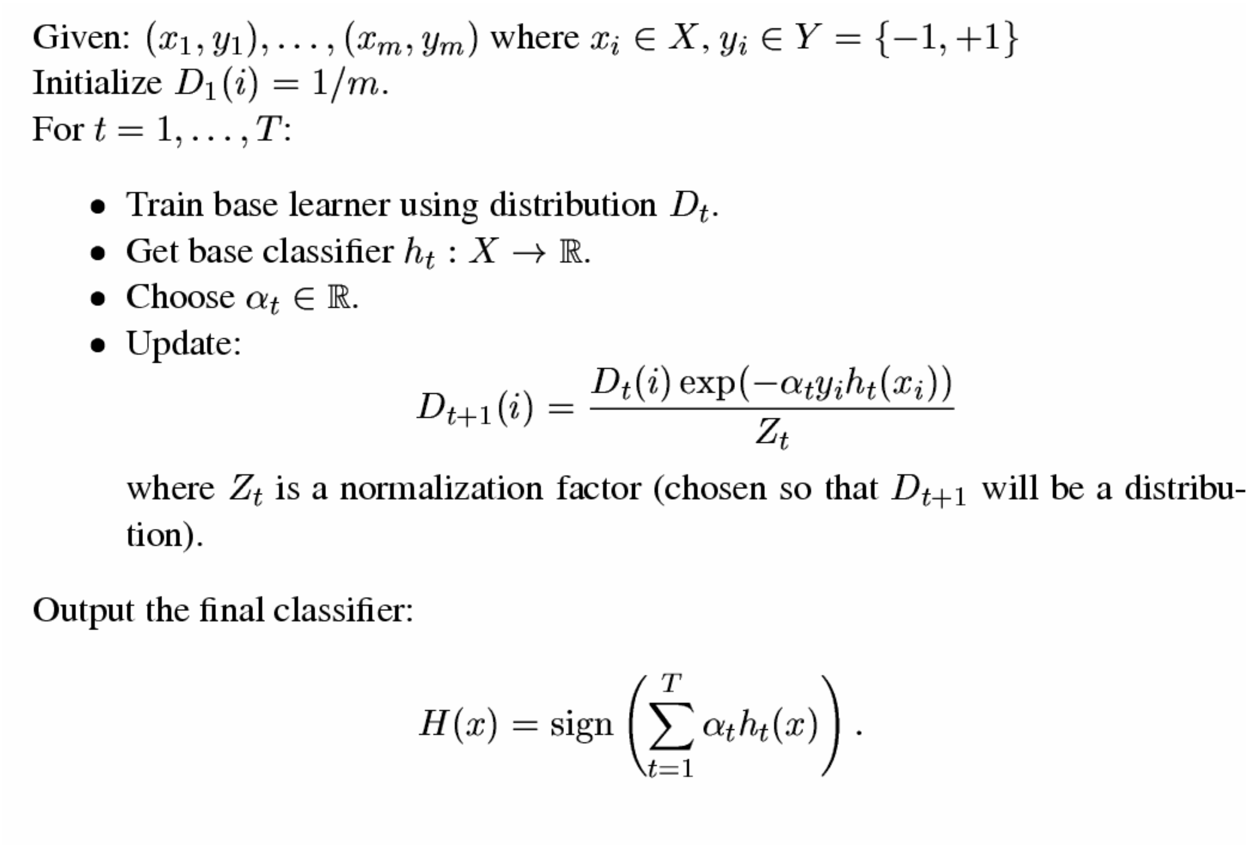
Note: As I am using Random integer generator for choosing the training data samples, the parameter values generated for above parameters vary by some buffer and hence exactly same results might not be generated but it should be almost equal in the range mentioned above. Except for some cases when random data chosen is very illogical and might give low results but this **can be verified by running the algorithm multiple times**.

Boosted Decision Trees

Introduction:

Boosting can be done on decision trees by incorporating weights to the classified labels depending on their previous accuracy. I have developed boosting on the Decision trees just like the Random forest algorithm.
The idea here is, given a weak learner (testing dataset), instead of just taking the vote of trained dataset directly, I run the weak learner recursively and reweight the already presents weights and then take vote from the learned classifier after reweighting it multiple times.
The process is as, on each iteration, weight each training example by how incorrectly it was classified by the previous learner. This algorithm works better because sometimes, some data points are more equal than others.

Algorithm:



Given: $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \in X, y_i \in Y = \{-1, +1\}$
Initialize $D_1(i) = 1/m$.
For $t = 1, \ldots, T$:

- Train base learner using distribution D_t.
- Get base classifier $h_t : X \to \mathbb{R}$.
- Choose $\alpha_t \in \mathbb{R}$.
- Update:

$$D_{t+1}(i) = \frac{D_t(i)\exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$

where Z_t is a normalization factor (chosen so that D_{t+1} will be a distribution).

Output the final classifier:

$$H(x) = \operatorname{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right).$$

Figure 1: The boosting algorithm AdaBoost.

Flow of Code:

readFiles () - Reads the txt files and puts the data into the ArrayList<ArrayList<Integer>> as dataset1 and dataset2. For string value "PRESENT" it adds 1, and for "ABSENT" it adds 0 of type double.

***Main ()* –** Divides the data into Training and Test part. I am taking 30% of data as test data and 70% as training data at first. It then processes the test and train data and creates the data matrices for each training and testing datasets. Then it Calls the getAllParameters () method.

getAllParameters (arguments) – Here, start with assigning an initial weight to the training dataset and then recursively use weighted learner to classify the points in testing dataset. This is the main function to start building the binary tree and calculating the accuracy, precision, recall and F-measure parameters. It is called once for each dataset.

createBinaryTree (arguments) – this method calls the createBinaryNode () method which first creates a root node for the binary tree and then recursively creates Binary tree nodes and their children and add these nodes to the root of the main binary tree.

CrossValidate ()

It performs the 10-fold cross validation on the data, by setting one part as test data, and rest as train data at each time. The data is divided in random parts, by adding the leftover data buffer into each partition. Then average values for each fold of rotation are calculated for accuracy, precision, recall and F-measure.

Pros:

1. It converts a group of "weak learners" together to form a "strong learner
2. It runs efficiently on large data bases
3. Weights assign importance depending how close data is to correctly classified data
4. It has methods for balancing error in class population unbalanced data sets

Cons:

1. Doesn't work for inputs that haven't been seen
2. Doesn't work well too many classes.

Results:

10 Fold Cross Validation Average Values for Boosted Decision Trees

Parameter	**Dataset 1**	**Dataset 2**
Accuracy	0.9307042253521126	0.7565217391304347
Precision	0.8983673469387755	0.6808510638297872
Recall	0.9175	0.7111111111111111
F-Measure	0.8978350515463918	0.6956521739130436

Note: As I am using Random integer generator for choosing the training data samples, the parameter values generated for above parameters vary by some buffer and hence exactly same results might not be generated but it should be almost equal in the range mentioned above. Except for some cases when random data chosen is very illogical and might give low results but this **can be verified by running the algorithm multiple times**.