CSE 601 Data Mining and Bioinformatics
Programming Assignment 3
Classification Algorithms

Group No 2

Arnav Ahire [5020 8006]
Ashwin Nikam [5020 7368]
Kedar Paranjape [5020 5932]

K-Nearest Neighbor

Algorithm

The K-nearest neighbor (k-NN) algorithm is a lazy learning algorithm that learns by analogy. That is, it compares a tuple with other training tuples that are most similar to it. Each tuple is a point in an *n*-dimensional space, where n is the number of attributes in the data set. The training dataset consists of tuples whose classes are known to be correct. To classify a new point, k-NN calculates *k* closest tuples to it in *n*-dimensional space using some distance metric. In our implementation, this metric is taken to be the Euclidean distance. Then, the new point is assigned a class label corresponding to the majority class label amongst these *k* training tuples.

Parameter Setting

The value of *k* determines the granularity of the boundary between predicted classes. This usually is data dependent, but generally, a higher value of *k* makes the algorithm robust against outliers. This is because we take more votes into account, but also blurs the boundaries between classes. A lower value of *k* will give crisper class boundaries but will also suffer from local minima. For binary classification, an odd value of *k* will allow for a majority to be calculated.

Pre processing

Data is typically normalized before feeding it to k-NN algorithm to prevent attributes with larger ranges from outweighing those with lower ranges (i.e when features are not scaled correctly with respect to their importance). Several forms of normalizations can be performed on the input data. A popular method is the Z-score normalization. If x is the value for the nth attribute, then the scaled attribute value(x') is obtained by:

$$\mathbf{x' = (x - mean_n) / std_n}$$

where $mean_n$ and std_n are the mean and standard deviation for the nth attribute respectively. While classifying a test tuple, the same normalization can be performed on it using the values for mean and standard deviation obtained earlier for the training data.

Handling continuous/categorical attributes

Since Euclidean distance is used as a distance metric, numerical attributes do not need any manipulation in terms of how they are represented. However, for categorical attributes, a simple method to calculate their distance is to assign 0 when the values for that attribute exactly match or 1 when they don't. Other, more granular distance schemes may also be used which would assign different weights amongst different values. For e.g. dist(green, red) = 0.6 while dist(green, blue) = 0.8 and so on. In our implementation, we have used the former (0/1 method)

Pros

- It is straightforward to implement and it skips the training phase.
- The distance metric is configurable and can be tweaked to get better results.

Cons

- Nearest Neighbor classifiers use distance as their defining metric and thus give equal weights to every attribute. Hence, their accuracy can drop significantly when the data set contains noisy or irrelevant attributes.
- It can be computationally expensive for large training sets because of the distance calculation and linear searching
- If k is very small then model is sensitive to noise points and if k is too large then it may include points from other classes.

Result Analysis

- knn is a lazy learner algorithm as we don't train the model.
- We also get the results very quickly as it doesn't involve a lot of processing.
- By having a look at the results we can clearly see that accuracy of knn for dataset1 is very high.
- In our algorithm the distance between two continuous attributes can be clearly defined using euclidean distance however distance between two different categorical attributes is always one.
- This might make a dataset having all continuous values more suitable to this algorithm for classification as shown from the results.

Cross Validation

K = 5

The results have been cross validated using 10 fold validation. The accuracy values for each fold have been listed below:

project3_dataset1.txt	project3_dataset2.txt
0.9642857142857143	0.6086956521739131
0.9642857142857143	0.782608695652174
0.9642857142857143	0.6304347826086957
0.9642857142857143	0.7391304347826086
0.9107142857142857	0.5652173913043478
0.9821428571428571	0.5869565217391305
1.0	0.717391304347826
0.9821428571428571	0.7391304347826086
0.9642857142857143	0.5217391304347826
0.9692307692307692	0.6666666666666666

Average metrics for the same over all the 10 folds are given below:

	project3_dataset1.txt	project3_dataset2.txt
Average Accuracy	0.966565934065934	0.6557971014492753
Average Precision	0.977994227994228	0.5131229799612153
Average Recall	0.9269071669071669	0.38737565066512436
Average F1-Measure	0.9504402610752484	0.4155907212249691

Decision Tree

Algorithm Flow

Decision trees are built for classification tasks where we need to classify a data point saying that it belongs to a particular class. We are given a training set on which we build our model which in this case in a decision tree. Then we use the test set in order to test the accuracy of the model and see whether it can predict the class of each tuple in the test set correctly. We implemented the decision tree algorithm called as CART which stands for Classification and Regression Trees.

The tree which is built is a binary tree where, at each node we first check if all the records belong to the same class label. If not, we try to find the best split at each column by computing impurity for each split in the column using the gini index. The best splits for each of these columns are taken and the one which gives lowest impurity among them is taken as the split for that node and the data is divided according to that split.
This process is carried on repeatedly until either we reach a point where all records belong to the same class or when all records have similar attribute values . Computing the best split has to be handled differently for categorical and continuous data. Let's have a look at it.

How to deal with categorical features

In case of categorical features first we are mapping these categories into numbers. So if there are two categories 'present' and 'absent', they become 1 and 0. This is basically done in order to convert the entire string numpy data matrix into a float data matrix so that future computations can be made easy. We also keep a record of which columns are categorical and which columns are continuous so it becomes easy to determine which data is categorical and which data is continuous. Then we find all the combinations of the available categories. For example, if the categories were 'Sunny', 'Cold' and 'Humid' they would first be mapped to 0, 1, 2 and then we would try to find all the possible combinations which are [0, 1] [0, 2] [1, 2] etc. They are considered to be splits. For each of these splits we create two sets. Then we traverse each row in the training data and check if the attribute value is in the split. For example, if we are computing impurity for split [1, 2] we check each row for that attribute value. If the value is 1 or 2 we add that row to set1 else set0.

Then we calculate gini index for both sets and then compute impurity for the split [1, 2]. We select the split which minimizes the impurity. This process is repeated for each of the split and thus we get the split which gives us the least impurity. We divide the data using this split. *handle_categorical_data()* method in our code is used to handle this.

How to deal with continuous features

In case of continuous features we take that specific continuous feature value of each row as a split and compute impurity using that split. So whichever rows have feature value less than the split are considered to be in set0 and whichever rows have a feature value >= split are considered to be in set1. Gini index is computed for both sets and then overall gini (impurity) is calculated for the split. We try to minimize this impurity by choosing the split which gives least impurity.
handle_numerical _data() method in our code is used to handle this.
In our algorithm, we sorted the entire matrix given to this method using the *argsort()* method such that the matrix is sorted according to the column of the feature value. Then we traverse through each row of the matrix and select that feature value as a split and the two sets are rows which are above and below the current row as we have sorted the matrix. Once we have our two sets we do the remaining method the same way as done for categorical data.

How to choose best features

For each feature we call the *handle_categorical_data()* or *handle_numerical_data()* method depending on the data. This gives us the split value and the corresponding gini value for the best split at that particular feature. Thus we get a list of split_values and gini_values for all the features in the data. We select the split value which corresponds to the least gini_value as we need to split the data in such a way that impurity is minimum. Thus data is split using this split_value and the entire process is repeated. In this way, we always choose the best feature which gives the least impurity at each node in the decision tree.

Post Processing

We can improve the accuracy of decision tree by using some post processing techniques like post-pruning. Here we trim the nodes of the decision tree in a bottom up fashion. If generalization error improves after trimming then replace the subtree by a leaf node whose class is determined by the majority of instances in the subtree. We tried implementing this in our algorithm however we didn’t get the desired results.

Pros

- Simple to understand and interpret.
- Pretty useful and accurate when the dataset is small.
- This model allows us to consider as many different consequences of a decision and weigh the tradeoffs of one decision against another.

Cons

- Tree creation becomes difficult and convoluted if there are a lot of uncertain values in the dataset.
- Decision trees are not that particularly good with handling continuous variables
- Decision tree suffers from instability. Slight change in the input can cause a completely different tree to be created. Hence it lacks robustness.
- Causes overfitting issue resulting in reduced accuracy. Needs pre-pruning and post-pruning to improve the accuracy of this classifier.

Result Analysis

- By observing the results we can clearly see that knn has a slightly better accuracy than decision tree.
- One major cause of this could be overfitting which leads to reduction in accuracy. Overfitting can be dealt with by preprocessing and postprocessing techniques like pruning.
- Moreover, decision trees are useful when dataset is quite small. In our case we are dealing with a comparatively larger dataset hence the reduction in accuracy, compared to knn.

Cross Validation

The accuracy at each fold, for the tree generated using training data at that fold is shown below. The accuracy has been computed using the testing data at that fold.

project3_dataset1.txt	**project3_dataset2.txt**
0.9285714285714286	0.6304347826086957
0.9285714285714286	0.7391304347826086
0.9285714285714286	0.6521739130434783

0.9642857142857143	0.6086956521739131
0.875	0.5434782608695652
0.9107142857142857	0.5
0.9285714285714286	0.5869565217391305
0.9464285714285714	0.782608695652174
0.8928571428571429	0.6739130434782609
0.8923076923076924	0.6458333333333334

The following table shows average accuracy, precision, recall and f1-measure for 10 fold cross validation.

	project3_dataset1.txt	**project3_dataset2.txt**
Average Accuracy	0.919587912088	0.636322463768
Average Precision	0.794755728253	0.4765317139
Average Recall	0.797880637881	0.489677398625
Average F1-Measure	0.794004708094	0.476358253748

Naive Bayes

Algorithm Flow

Naive Bayes is a statistical classifier which is based on Bayes Theorem. This classifier predicts the class membership probabilities such as the probability that a tuple belongs to a particular class. This algorithm assumes that all the attributes of a tuple are independent of each other and have no correlation whatsoever. This assumption is called class-conditional independence.

Let *H* be the hypothesis that a tuple *X* belongs to a class label *C.* For classification, we look for the probability that the tuple *X* belongs to class *C* given that we know the attributes of that tuple. This is also called as the *posterior probability* and is given by *P(H|X).*

Terms:

- *P(H|X)* : Probability of hypothesis *H* holding true that a tuple *X* belongs to class *C* given that we know the attributes of *X*. (Posterior probability) (a.k.a posteriori probability of *H* conditioned on *X*).
- *P(H) :* It is the prior probability of hypothesis H holding true which is independent of the data tuples.
- *P(X|H)* : It is the probability that there exists a tuple *X* such that it satisfies the hypothesis *H* of belonging to class *C*. (Posterior probability of *X* conditioned on *H*)
- *P(X) :* It is the prior probability of existence of tuple *X*.

This is the formula we are going to use to perform classification:

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

Implementation:

- In the above formula *P(X)* is constant for all the classes and hence we have ignored this term in our implementation.
- First we divide the dataset into training data and test data.
- For a tuple in the training data with class label *Ci*, if we come across a numerical column(attribute), we calculate mean and variance for that column of the dataset. We do not perform any calculations for the tuple in case of a categorical column.

- The posterior probability for each tuple *X* i.e *P(X|Ci)* or *P(X|H)* in the test data is calculated depending on the attribute type. For all the numerical attributes we assume normal distribution and calculate probability density given by the formula:

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

  Where:
  μ = Mean
  σ = Standard Deviation
  σ^2 = Variance
  We use the mean and standard deviation values that we already calculated for each column, to calculate this probability. If we come across a categorical column, we calculate the number of times the value in that column has appeared in the dataset for the *Ci* and divide by the count of total number of *Ci*s in the dataset. We ignore the last column which represents the class label for our training data .
- The posterior probability *P(X|Ci)* is the multiplication of the calculated probabilities over all the attributes to the prior probability *P(Ci)* or *P(H).*

How to handle categorical values

As explained in the algorithm, we handle categorical columns and numerical columns differently. In case of training data, we have maintained a dictionary that keeps a track of our class labels. If we find that the column is categorical we simply append the value as keyword ‘Categorical’ to the key of our dictionary which is the class label corresponding to our tuple. In case of test data, we divide the count of occurrence of our categorical value for class label *Ci* by the total count of *Ci* present in the dataset.

How to handle continuous values

In case of training data, we have maintained a dictionary that keeps track of our class labels. If we find that the column is numerical, we calculate the mean and standard deviation over all the values in that column and append the calculated mean and standard deviation to the key (corresponding class label) of the dictionary. This calculation is done only once per column. In case of test data, for each tuple, we calculate the probability density for each column of that tuple using the mean and standard deviation values that were already stored in the dictionary for that column.

Pros

- Simple and easy to understand and implement.
- If the conditional independence assumption actually holds true, then this classifier converges faster than other classifiers, hence will require less training data.
- Handles continuous as well as discrete data.
- Makes probabilistic predictions.
- Efficient when dealing with large databases.

Cons

- This classifier doesn't perform well if the dataset has correlated attributes. Hence called 'Naive'.
- Faces a *'zero-probability'* issue if you do not have a particular attribute value present in the training data, it will give a zero probability while calculating the posterior probability on the tuple that has that attribute value. This drawback can be handled using *'Laplacian Correction'* however.

Result Analysis

- As we can observe, this algorithm is fairly easy to implement and is also quite effective since it gives a very high accuracy for the project3_dataset2.txt.
- Each time the algorithm assumes from the start that all attributes are unrelated.
- Hence in case of unrelated attributes this algorithm performs very effectively.
- project3_dataset2.txt is the best example whose accuracies explain the effectiveness of this classifier.

Cross Validation

We performed 10 fold cross validation where at each fold we sampled k times with replacement from the training data set to create k models. Every test data is passed through all k models. The overall accuracy for test data at each of the 10 folds is shown below:

project3_dataset1.txt	**project3_dataset2.txt**
0.9464285714285714	0.6739130434782609

0.9285714285714286	0.7391304347826086
0.9642857142857143	0.782608695652174
0.9107142857142857	0.6956521739130435
0.9107142857142857	0.6304347826086957
0.9642857142857143	0.6086956521739131
0.9642857142857143	0.8478260869565217
0.9642857142857143	0.782608695652174
0.9285714285714286	0.6086956521739131
0.8923076923076924	0.6875

The average metrics for accuracy, precision, recall and f1-measure for 10 folds on both datasets are given below as follows:

	project3_dataset1.txt	**project3_dataset2.txt**
Average Accuracy	0.937445054945	0.705706521739
Average Precision	0.739306147196	0.575332325913
Average Recall	0.715966255966	0.620826425037
Average F1-Measure	0.726899577387	0.589489749977

Random Forests

Algorithm Flow

Random Forests is a part of ensemble learning wherein we compute a set of *t* classifiers for *t* subsets of training data. The testing phase involves a voting by each of these classifiers to determine the class of the training data and depending on the majority the class of the test data is chosen. We used bootstrap sampling with replacement in order to sample data from the training set each time for creating a classifier. Thus the data used for creating a classifier each time may contain duplicates. A hyperparameter ***t*** is chosen and that many classifiers or trees are constructed where each time, random samples are chosen from the training data with replacement for construction of the tree. Thus each tree will be different in some way from the other as each of them have been built using different data. Another important hyperparameter in this case is ***m*** which needs to be set at a predefined value which has to be quite low (For example, 20% of the number of features in the data). We have chosen m to be the square root of total no. of features in the data.

During construction of the trees, at each node we choose *m* random features and find the best split among these m random features. A continuous feature can be chosen multiple times however a categorical feature can be chosen only once along that path starting from root to the current node. Stopping conditions for a tree in random forests are when a node becomes too small (<= 3 records) or when height of the tree exceeds the number of features.

After construction of *t* such trees we run the same test data through each of these 't' trees and gather votes from them. For example, if there are 5 trees and 3 of them classify a sample test point as 1 while 2 of them classify the same point as 0, then we assign the class label 1 to that test data point due to majority. This is the concept of ensemble learning.

Parameter Setting

The parameter '*t*' is the number of decision trees that the forest will contain. A higher value reduces variability in classification and thereby reduces overfitting. However this imposes a runtime penalty.

The parameter *'m'* refers to number of attributes to randomly choose from while computing the best split at each node. Usually, *'m'* has to be a very small value (20% of the total no. of attributes)

Pros

- Handles the overfitting problem of decision trees since here averaging of several trees is performed.
- It is one of the most accurate learning algorithms producing a highly accurate classifier.
- It handles large datasets very well since we do not consider all the attributes while creating random trees.
- It can handle several input variables without having to delete them.

Cons

- This model is slower because it takes into account several trees to classify the test data.

Result Analysis

- From the results, it is evident that random forests outperformed decision trees. This is because of the fact that decision trees tend to overfit to their training sets. Thus, an ensemble of forests initialized to trees constructed using a random set of attributes corrects for these overfitting tendencies.
- Since we use bootstrapping (sample training data with replacement for each classifier), the variance of the model is reduced without increasing the bias. This means that even if the predictions of a single tree are sensitive to noise, the average of many trees is not (as long as the trees are uncorrelated).

Cross Validation

We performed 10 fold cross validation where at each fold we sampled t times with replacement from the training data set to create t trees. Every test data is passed through all t trees, and based on the majority of votes of the trees a class is assigned to that test data. The overall accuracy for test data at each of the 10 folds is shown below:

project3_dataset1.txt	**project3_dataset2.txt**
0.9642857142857143	0.5652173913043478

0.9464285714285714	0.8043478260869565
0.9642857142857143	0.6739130434782609
0.9464285714285714	0.6739130434782609
0.9107142857142857	0.6521739130434783
0.9642857142857143	0.5217391304347826
0.9642857142857143	0.6956521739130435
0.9821428571428571	0.8260869565217391
0.9464285714285714	0.6304347826086957
0.8923076923076924	0.6666666666666666

The average metrics for accuracy, precision, recall and f1-measure for 10 folds on both datasets are given below as follows:

	project3_dataset1.txt	**project3_dataset2.txt**
Average Accuracy	0.948159340659	0.671014492754
Average Precision	0.943750693751	0.540182595183
Average Recall	0.91865985866	0.40585919928
Average F1-Measure	0.929384597715	0.450826096824

These values would change each time slightly depending upon the random samples taken.

Boosting (AdaBoost)

Algorithm Flow

Boosting is a type of ensemble learning algorithm which strives to improve the learners by focusing on areas where the system isn't performing well. AdaBoost is the algorithm we use in this project which constructs a strong classifier as a linear combination of weak classifiers. We need to set a hyperparameter ***t*** which corresponds to the number of trees we want to build at each fold. At each fold we compute a test data set and train data set. We also maintain a weight vector of all the train data which is set as 1/n (n = no. of points in the training set) for all points at the start. This train data set is used for bootstrap sampling with replacement to generate a sample data set based on the weights. Points having a higher weight are more likely to get selected. We then create a tree using this sample data. The tree creation is done using the same algorithm as decision tree. Now, the train data is used in order to test the accuracy of the model and error is computed by summation of the weights of misclassified points.

$$\varepsilon_i = \frac{\sum_{j=1}^{N} w_j \delta(C_i(x_j) \neq y_j)}{\sum_{j=1}^{N} w_j}$$

If the error is greater than 0.5 we choose a new sample and again create a model. After receiving an error less than 0.5 we find the alpha parameter of the model using the formula

$$\alpha_i = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_i}{\varepsilon_i} \right)$$

We then update the weights of each point based on the classification from the model. Weights are updated using the formula

$$w_j^{(i+1)} = \frac{w_j^{(i)} \exp\left(- \alpha_i y_j C_i(x_j)\right)}{Z^{(i)}}$$

Our aim is to increase the weights of the misclassified points and decrease the weights of the correctly classified points. *math.exp()* helps us achieve that wherein if the point is correctly classified the value inside *math.exp()* is positive which helps in reducing the value and vice versa.
After the weights have been updated we normalize the weights so that they add up to 1. This entire procedure is done upto t trees in a single fold. At each iteration of t we boost the weights of misclassified points and decrease the weights of correctly classified once thus our models start becoming more accurate.

In the classification part we get votes of each model however, we also consider their alpha values. For example if a model A has alpha 1.1 and has voted for the point to be in class 0 while other two models B and C having alpha values 0.5 and 0.4 respectively have voted for the point to be in class 1, the point would still be classified as 0 because model A had alpha value greater than that of B and C combined.

Parameter Setting

Here we set the parameter '*t*' that represents the number of trees to be used. Higher the '*t*' value, reduced will be the variability and this will reduce the chances of overfitting.

Pros

- We don't need to specify any parameters at the start except '*t*' which represents number of trees at each fold.
- It is fast in execution.

Cons

- Sensitive to noisy data and outliers.

Result Analysis

- Boosting is an ensemble learning algorithm like random forest.
- It performs better than decision tree which can be seen from the results below.
- This is mainly because boosting tends to increase weights of misclassified points and decrease weights of correctly classified points each time.
- So the next time a random sample is taken from training data, probability of the misclassified points appearing in the sample are higher.
- This helps improve accuracy of boosting compared to decision tree.

Cross Validation

t = 5 (5 trees are generated at each fold)

We applied 10 fold cross validation in order to test the accuracy of this model for both datasets. In each fold we have a different train and test data. The train data is used to sample points which have weights assigned to it. At each fold we are creating 5 trees thus 5 samplings are done on the same train data corresponding to that fold. The accuracies of the model for the 10 folds on both datasets are given below as follows:

project3_dataset1.txt	**project3_dataset2.txt**
0.9464285714285714	0.6521739130434783
0.9642857142857143	0.6739130434782609
0.9821428571428571	0.717391304347826
0.9464285714285714	0.6521739130434783
0.9285714285714286	0.5434782608695652
0.9464285714285714	0.5
0.9464285714285714	0.6521739130434783
0.9464285714285714	0.7391304347826086
0.9285714285714286	0.6086956521739131
0.9538461538461539	0.6666666666666666

The average accuracy, precision, recall and f1-measure of all the folds are given below.

	project3_dataset1.txt	**project3_dataset2.txt**
Average Accuracy	0.948956043956	0.640579710145
Average Precision	0.944552762053	0.472048539696
Average Recall	0.919909349909	0.431981395797
Average F1-Measure	0.93047122931	0.439813946773

These values would change each time slightly depending upon the random samples taken.