CSE 601 : Data Mining and Bioinformatics

Project 3 Classification Report

rohanhem | 50291746

nd34 | 50289820

K-Nearest Neighbors:

Implementation Flow:

- Since the data has both continuous and categorical features, first the categorical feature columns are identified and all the unique feature values in each of those columns are replaced by integers corresponding to the number of unique feature values in those columns, starting from 0.
- 10-fold cross validation is performed to get a good estimate of the accuracy measures of the algorithm. At each fold, we divide our data into train data and test data. Every test sample from this test set is normalized along the train data. A temporary data is created with the rows of train data and an additional row of the test sample without the last column. From each column of this temporary data, mean along that column is subtracted and it is divided by standard deviation along that column. Then this temporary data is again separated into train data and test sample with their last columns.
- A test point/sample is classified by calculating the distance between this sample and all other train data points. We measure distance metric differently for continuous feature columns and categorical feature columns. For continuous feature columns, we use Euclidean distance, and for categorical feature columns, we get dissimilarity using Hamming distance. We sum these distances to get the distance between a test point and a train point.
- Once we have all the distances, we use argsort to get the indices of the k closest train points and assign the class that the majority of the k train points belong to, to the test point. This is repeated for all the test points in every fold.

Dataset 1

Accuracy: 96.663533834586 47 %

Precision: 98.54219948849105 %

Recall: 92.58877591865794 %

F-1 Measure: 95.29281749376221 %

Dataset 2

Accuracy: 69.26919518963922 %

Precision: 64.09523809523809 %

Recall: 29.613050691998062 %

F-1 Measure: 39.26744596049417 %

Parameter Settings: We varied k values to check which one gives the maximum accuracy. We ended up with selecting k=9 for dataset1 and k=40 for dataset2 to get good accuracy values. **I**f the k value is very small, the classification is affected by noise. If it is very large, the neighborhood may include points from other classes as well.

Pros:

- Scales well with complexity of attributes as we have the luxury of choosing distance functions.
- Fairly simple to implement.
- Can be modified and scaled quickly as per needed if there are some changes to the requirements/dataset.

Cons:

- The flip side of choice of distance functions is the increased computation complexity to find distances and the minimum.
- Storage requirement is also on the higher side as we need to store the distance matrices.
- A small error in defining distance functions can induce incorrect results.
- If the dataset is large, the algorithm can be slow/inefficient in its performance.

Conclusion:

- Absence of training makes the algorithm quick and cuts down on processing.
- We observe that for the first dataset we get a very high accuracy which cannot be replicated for the second dataset as the first one (with continuous values) is more suitable for the algorithm than the other.
- As we go on to other algorithms, we will observe that even after accounting for the drawbacks, this algorithm still performs the best.

Naïve Bayes:

Implementation Flow:

- Since the data has both continuous and categorical features, first the categorical feature columns are identified and all the unique feature values in each of those columns are replaced by integers corresponding to the number of unique feature values in those columns, starting from 0. 10-fold cross validation is performed to get a good estimate of the accuracy measures of the algorithm. At each fold, we divide our data into train data and test data.

- We collect the training samples in the respective classes separately into two lists. Then we calculate prior probabilities of the classes by dividing the number of unique classes in the last column by the length of train data.

- We handle continuous and categorical features separately. The data is separated into continuous and categorical so that both of them can be handled separately. If we come across a categorical column, we calculate the number of times the value in that column has appeared in the dataset for each of the classes and divide by the total number of those labels in the dataset.

- For continuous data segment, we calculate column-wise mean and standard deviation in order to get Gaussian distribution. Similarly for categorical, we calculate prior and posterior probability. We find discrete probability separately using count on posterior and prior probability of a particular test sample.

- In testing phase, for each test sample, we calculate conditional probability as in Bayes theorem for every class and we multiply continuous and categorical probability for getting total class-wise probability.

- Class with the higher probability will be assigned to each test point at every fold.

Dataset 1

Accuracy: 93.49310776942357 %

Precision: 88.09442634660027 %

Recall: 95.25948926328434 %

F-1 Measure: 91.32451244200851 %

Dataset 2

Accuracy: 69.69935245143385 %

Precision: 56.08259193681189 %

Recall: 66.82484474589738 %

F-1 Measure: 60.344668696175 16 %

Zero probability: Here, if the posterior probability is zero then we use Laplacian correction, but we didn't code it in our system.

Pros:

- Independence of labels makes the algorithm easy to implement.
- Fast performance and simple in structure.
- Algorithm is quite transparent in the sense that it is easy to see which attributes are influencing the outcomes.

Cons

- Accuracy of prediction can fall off depending on the dataset, relative to other algorithms.
- Reliance on feature independence can lead to some wrong assumptions and relations between features can induce errors.
- Zero posterior probability is a possibility.

Conclusion:

- The algorithm is fairly easy to implement and gives a high accuracy result for the first dataset.
- We see a big drop off in the second dataset, though a slightly better overall performance than knn if we compare the recall.

Decision Tree:

Implementation Flow:

- We first deal with the categorical features by assigning integers to all the unique values and then we start the cross validation fold by fold (upto 10)
- A decision tree is built at every fold using best node splits and the training data and then we use Gini index to determine the best split.
- For each tree, by calculating impurity for each split in the column, we find the lowest impurity which gives us the best split. This is carried out for all the columns and the data is divided accordingly.
- This continues until all records have been converged to a single class or when they reach similar attribute values.
- For division, we place the rows with feature value less than the split to the left and those with greater value to the right. For both sets, we calculate the Gini index and then overall impurity is calculated. We try to minimize this by choosing the split with least impurity.
- This process is then repeated and we select the best features with least impurity at every node in the tree. The whole tree is then built.
- We then use the test data to classify the samples by traversing the tree that we have just built.

Dataset 1

Accuracy: 92.08646616541353 %
Precision: 89.07526002262844 %
Recall: 90.07324122219757 %
F-1 Measure: 89.33253545104334 %

Dataset 2

Accuracy: 64.8936170212766 %
Precision: 49.62631456091209 %
Recall: 50.306331826068664 %
F-1 Measure: 49.15735680889418 5 %

Pros:

- Categorical data can be handled in one go, normalization not required.
- Missing values do not largely affect the outcome.
- Gives good accuracy on par with other algorithms.
- Can be easily combined with other techniques.

Cons:

- Larger datasets may make the algorithm quite complex and slow.
- A small change in input data can induce a large corresponding change at the output.
- Predicting continuous value can be a problem.

Conclusion:

- The algorithm performs fairly well though there could be performance dropoff due to overfitting.
- This could be overcome using techniques like pruning.
- Out dataset is relatively large, thus making comparisons even more unfavourable to algorithms such as knn as this algorithm is more suited for smaller datasets.

Random Forest:

Implementation Flow:

- Since the data has both continuous and categorical features, first the categorical feature columns are identified and all the unique feature values in each of those columns are replaced by integers corresponding to the number of unique feature values in those columns, starting from 0. 10-fold cross validation is performed to get a good estimate of the accuracy measures of the algorithm. At each fold, we divide our data into train data and test data.
- Bagging is used to sample the data (with replacement). Then, the data is used to construct a classifier. We use hyperparameter 'ntrees' to denote the number of trees chosen, which indicates how many classifiers are created. Note that random samples are chosen with replacement for construction of the tree.
- What this is useful for, is that we get distinct trees as all have been constructed using different data.
- The hyperparameter rand_feat denotes the number of features divided by 3.
- After construction of ntrees such trees we run the same test data through each of these 'ntrees' trees and gather votes from them and class with the majority votes is chosen.

Dataset 1

Accuracy: 95.6015037593985 %
Precision: 97.38870851370851 %
Recall: 90.22390631915422 %
F-1 Measure: 93.634841866714678 %

Dataset 2

Accuracy: 63.8436632747456 %
Precision: 45.71015096015096 %
Recall: 32.633464488727765 %
F-1 Measure: 37.52976833911301 %

The parameter ‘ntrees’ 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 ‘rand_feat’ refers to number of attributes to randomly choose from while computing the best split at each node.

Pros:

- Fast performance and one of the most accurate algorithms for classification.
- Can be used for dimensionality reduction as it can handle multi-dimensional data much better than many other algorithms.

Cons:

- Can be difficult to read if compared to models like decision tree.
- Can eat up a lot of memory if the input dataset is large.
- Overfitting can be an issue.
- Number of trees has to be decided beforehand.

Conclusion:

- We can see that Random Forest has performed better than Decision Trees and this indicates an overfitting problem with decision trees, which gets corrected in Random Forest as an ensemble of trees constructed using random attrributes eliminates overfitting.
- Bootstrapping helps in reducing the variance without affected the bias, thus if a single tree gets adversely affected by noise, the average of many trees keeps the problem at a minimum.

Final summary:

Dataset 1

Algorithm	Accuracy	Precision	Recall	F-measure
KNN	96.66	98.54	92.59	95.29
Naive Bayes	93.49	88.09	95.26	91.32
Decision Tree	92.09	89.08	90.07	89.33
Random Forest	95.60	97.39	90.22	93.64

Dataset 2

Algorithm	Accuracy	Precision	Recall	F-measure
KNN	69.27	64.09	29.61	39.26
Naive Bayes	69.70	58.08	66.82	60.34
Decision Tree	64.89	49.63	50.31	49.16
Random Forest	63.84	45.70	32.63	37.53

Kaggle Competition:

- Our approach towards improving the accuracy of base classifiers such as Decision Tree, Random Forest, AdaBoost, Naive Bayes, K-NN was to use an Bagging Classifier and use a combination of several base_estimators and a ParameterGrid.

```
grid = ParameterGrid({"max_samples": [0.5, 1.0],
                      "max_features": [1,2,4],
                      "bootstrap": [True, False],
                      "bootstrap_features": [True, False]
                      })
```

- We used n_estimators=100 for Adaboost and Random Forest and n_neighbors=1 for K-NN.

```
for params in grid:
        y_pred = BaggingClassifier(base_estimator=base_estimator,
                    random_state=0,
                    **params).fit(data, labels).predict(data)
```

- If y_pred and labels matched, we used that classifier to get the predictions for the test data,

```
        y_pred = BaggingClassifier(base_estimator=base_estimator,
                    random_state=0,
                    **params).fit(data, labels).predict(test)
```

- Ultimately, we used the labels that were predicted by majority of the predictions to get the best accuracy.