STATE UNIVERSITY AT BUFFALO

CSE 601 – Data Mining and Bioinformatics

PROJECT 3: CLASSIFICATION ALGORITHMS

NEAREST NEIGHBOR, DECISION TREE, AND NAÏVE BAYES

12/02/2017

Group 15:

Amaan Akhtarali Modak	- 50206525	- amaanakh
Arshdeep Gill	- 50139190	- asgill3
Kamalakannan Kumar	- 50205671	– kkumar2

1 k-Nearest Neighbor

1.1 ALGORITHM DESCRIPTION

kNN algorithm basically works by taking vote of the nearest neighbor of the particular element to assign a label for that element. kNN is non parametric meaning that it does not make any assumptions on the underlying data distribution

- we are given some data points for training and also a new unlabeled data for testing
- If x and y are sufficiently close, then we can assume that probability that x and y belong to same class is fairly same – Then by decision theory, x and y have the same class.

There is a non-existent or minimal training phase but a costly testing phase. The cost is in terms of both time and memory. More time might be needed as in the worst case, all data points might take point in decision. More memory is needed as we need to store all training data.

We are also given a single number "k". This number decides how many neighbors (where neighbors is defined based on the distance metric) influence the classification. This is usually an odd number if the number of classes is 2.

- Typically k is odd when the number of classes is 2:
  - Let's say k = 5 and there are 3 instances of C1 and 2 instances of C2. In this case , KNN says that new point has to labeled as C1 as it forms the majority. We follow a similar argument when there are multiple classes.
  - One of the straightforward extension is not to give 1 vote to all the neighbors. A very common thing to do is *weighted kNN* where each point has a weight which is typically calculated using its distance
  - Under inverse distance weighting, each point has a weight equal to the inverse of its distance to the point. This means that neighboring points have a higher vote than the farther points.
  - It is obvious that accuracy might increase when you increase k but computation cost also increases.
  - A small value of k means that noise will have a higher influence on the result.
  - A large value make it computationally expensive and kinda defeats the basic philosophy behind KNN (that points that are near might have similar densities or classes) .A simple approach to select k is set $k = \sqrt{n}$. But drawing classification boundaries for the dataset we can say that classification boundaries induced by for example k=1 are much more complicated than k= 15.
  - if we have a very large data set but we choose K to be too small, then we will still run the risk of overfitting. For example, for K = 3 and a really large data set, there's a reasonable chance that there will be two noisy data points that are close enough to each other to outvote the correct data points in some region. On the other hand, if we choose K to be too large, then we may end up smoothing things out too much and eliminating some important details in the distribution.
  - Basically need to choose K large enough to avoid overfitting, but small enough to avoid oversimplifying the distribution

1.2 IMPLEMENTATION of kNN algorithm

- Method name Dataset takes in two parameters, the training dataset and the testing dataset.
- Pandas dataframe made for the above two datasets
- get_numeric_data().columns function used to get the numeric data column numbers from the original test and train dataframe
- Both Dataframes converted to matrix using .as_matric function
- Conversion from string to float of all numeric values of training as well as test set
- Normalization done for all the values in both the array by using zscore normalization: (element-mean)/standard deviation
- Splitting: We use :
    - testRows= range(int(split*len(array_data)),int((split+.1)*len(array_data)))
    - All the rows of the array_data not belonging in testrows become the training set
        - Example if we make split of 90% training and 10% test then split=.1
- Set k value, k is the nearest neighbor to the test set
- Begin a for loop which iterate over the length of test set
    - Within above for loop we make nested for loop which iterate over the length of training set for every test set
    - For each test set we find the distance for all training sets to that test set
    - Make a list name compare_euc_sorted and append all the euclidean distance calculated for all training sets to particular test set
    - Sort the list with euc distances in increasing order inside that list by using sorted function
    - Sorting was done to pick k number of least euclidean distances from the aforementioned list
    - Later we loop over the k values stored to find out the majority label and that label becomes the label for the particular row of test set.
    - Same way we calculate label for all the training set.
    - Later we calculate the Accuracy, prediction, Recall, and F_1 measure values based on the outcome of the the label predictions of our test set and comparing it to the ground truth. This gives us the Truth table or the True positive, True negative, False positive and False negative values.
- Accuracy is calculated as follows: (TP+TN)/ (TP+TN+FP+FN)
    - It is simply the ratio of correctly predicted observations.
- Recall is calculated as follows: TP / (TP + p-FN)
    - Recall is also known as sensitivity or true positive rate. It's the ratio of correctly predicted positive events.
- Precision is calculated as follows: TP / (TP + FP)
    - Precision looks at the ratio of correct positive observations**.**
- F_1Measure is calculated as follows: ((2 * precision * recall) / (precision + recall))
    - The F1 Score is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. Intuitively it is not as easy to understand as accuracy, but F1 is usually more useful than accuracy, especially if you have an uneven class distribution. It works best if false positives and false negatives have similar cost. If the cost of false positives and false negatives are very different, it's better to look at both Precision and Recall.

Complexity of knn: $O(n^2)$ as have to go through nearest neighbours for every test set.

1.3 RESULTS OBTAINED

Results of knn for k=3:

For dataset1: running on

- Accuracy: 96.14
- Precision: 97.875
- Recall: 91.557
- F1_measure: 94.41

For Dataset2:

- Accuracy: 66.26
- Precision: 54.44
- Recall: 32.14
- F1_measure: 37.8895

1.4 PROS AND CONS

Advantages:

- Very easy to understand and implement. A k-NN implementation does not require much code and can be a quick and simple way to begin machine learning datasets.
- Does not assume any probability distributions on the input data. This can come in handy for inputs where the probability distribution is unknown and is therefore robust.
- Can quickly respond to changes in input. k-NN employs lazy learning, which generalizes during testing--this allows it to change during real-time use.

Disadvantages:

- Sensitive to localized data. Since k-NN gets all of its information from the input's neighbors, localized anomalies affect outcomes significantly, rather than for an algorithm that uses a generalized view of the data.
- Computation time. Lazy learning requires that most of k-NN's computation be done during testing, rather than during training. This can be an issue for large datasets.
- Normalization. If one type of category occurs much more than another, classifying an input will be more biased towards that one category (since it is more likely to be neighbors with the input). This can be mitigated by applying a lower weight to more common categories and a higher weight to less common categories; however, this can still cause errors near decision boundaries.
- Dimensions. In the case of many dimensions, inputs can commonly be "close" to many data points. This reduces the effectiveness of k-NN, since the algorithm relies on a correlation between closeness and similarity. One workaround for this issue is dimension reduction, which reduces the number of working variable dimensions (but can lose variable trends in the process).

Other Examples where kNN can be used:

- You can consider handwriting detection as a rudimentary nearest neighbor problem
- Gene expression
- Protein-Protein interaction and 3D structure prediction

2 DECISION TREES

2.1 ALGORITHM DESCRIPTION

Decision Tree is a supervised learning method for classification. The goal is to create a tree-structured model that predicts the class of a test sample by learning decision rules inferred from the data features. The decision rules are represented as internal nodes in the tree. Each internal node denotes a test on an attribute, each branch denotes the outcome of a test, and each leaf node holds a class label. When a new test sample needs to be classified, it goes down the decision tree from root to a leaf node according to the rules stored in internal nodes and the label stored in the leaf node determines its class label.

Decision tree builds classification or regression models in the form of a tree structure. It breaks down a dataset into smaller and smaller subsets while at the same time an associated decision tree is incrementally developed. The result is a tree with **decision nodes** and **leaf nodes**. A decision node (e.g., Outlook) has two or more branches (e.g., Sunny, Overcast and Rainy). Leaf node (e.g., Play) represents a classification or decision. The topmost decision node in a tree, which corresponds to the best predictor, is called the **root node**. Decision trees can handle both categorical and numerical data.

1: Sample Decision Tree Formulation

2.2 IMPLEMENTATION

Preprocessing: Before implementing the actual decision tree algorithm, we needed to preprocess the data and normalize the data to handle both categorical and continuous features. In the case of categorical features, it would be fairly straightforward each category to be selected as an attribute, but this approach is not feasible for continuous data. For this purpose, we implement normalization of data to convert each continuous attribute into a number of categories using binning. For continuous attributes, we discretized each of them into k bins. A sample's feature is labeled 0 if it's in the 1st bin, 1 if it's in the 2nd bin, 2 if it's in the 3rd bin and so on so forth, until k-1 if it's in the kth bin. After preprocessing, each feature will be a floating-point value.

Key Function Usage: We designed three key functions to finish the pipeline of decision tree classification, as follows:

1. build_tree(): build the tree structure.
2. information_gain(): calculate GINI to find the attribute with the highest entropy which will then be selected as the best choice.
3. classify_record(): classify new test samples according to the learned decision tree.

The best feature that is to be selected is found by calculating the GINI value (entropy), and the feature with the highest entropy value is selected as the best feature for that iteration. In our case, we are performing multi-way classification such that each node can be split into more than two branches as opposed to binary splitting of the tree.

2.3 PROS AND CONS

Advantages:

- Decision tree is easy to interpret, because it can be seen as a set of decision rules.
- At each internal node, decision tree will choose the feature that yields the lowest children's impurity, so it has the ability of selecting the most discriminatory features.
- Handling both continuous and discrete data.

Disadvantages:

- Need to discrete data for some particular construction algorithm.
- When dataset is small, it may yield large errors.
- Without pruning, the full tree can be complicated and overfitting.

2.4 RESULTS OBTAINED

Performance measures obtained when number of bins is set to 10:

Results for project3 dataset1.txt are:

Accuracy: 91.22807017543859
Precision: 87.5
Recall: 91.30434782608695
F-Measure: 89.36170212765957

Results for project3 dataset2.txt are:

Accuracy: 70.21276595744681
Precision: 52.63157894736842
Recall: 66.66666666666666
F-Measure: 58.82352941176471

3 DECISION TREE WITH RANDOM FOREST

3.1 ALGORITHM DESCRIPTION

Decision tree with random forest is an ensemble learning method for classification. Random forests can be built using bagging with random attribute selection at each splitting node. For bagging, we randomly picked the same number of samples as the training dataset with replacement and this is one round. We repeated the process for n times and each round will generate a decision tree. We will get n decision trees in the end and the class of a new test sample is determined by the equally weighted voting among all learned decision trees.

The procedure for training a random forest is as follows:

1. At the current node, randomly select *p* features from available features *D*. The number of features *p* is usually much smaller than the total number of features *D*.
2. Compute the best split point for tree *k* using the specified splitting metric (Gini Impurity, Information Gain, etc.) and split the current node into daughter nodes and reduce the number of features *D* from this node on.
3. Repeat steps 1 to 2 until either a maximum tree depth *l* has been reached or the splitting metric reaches some extrema.
4. Repeat steps 1 to 3 for each tree *k* in the forest.
5. Vote or aggregate on the output of each tree in the forest.

Compared with single decision trees, random forests split by selecting multiple feature variables instead of single features variables at each split point. Intuitively, the variable selection properties of decision trees can be drastically improved using this feature bagging procedure. Typically, the number of trees *k* is large, to the order of hundreds to thousands for large datasets with many features.

3.2 PROS AND CONS

Advantages:

- Tend to be more accurate.
- No distribution assumptions.
- Robust against outliers.
- Because random forests consider many fewer attributes for each split, they are efficient on very large databases.

Disadvantages:

- Less efficient sampling compared to boosting

3.3 RESULTS OBTAINED

Performance measure obtained for 10 fold cross validation average scores, number of bins is set as 5, and number of trees is set as 15.

Results for project3 dataset1.txt are:

Accuracy: 93.32161687170475
Precision: 95.31250000000001
Recall: 86.32075471698114
F-Measure: 90.5940594059406

Results for project3 dataset2.txt are:

Accuracy: 64.5021645021645
Precision: 42.30769230769231
Recall: 06.875
F-Measure: 11.827956989247312

Complexity of Decision tree is (mnlogn) m is amount of columns, n is the number of rows, logn as its splitting.

4 DECISION TREE WITH BOOSTING

4.1 ALGORITHM DESCRIPTION

Decision tree with random forest is also an ensemble learning method for classification. The main differences between boosting and Random forest is boosting does not have random feature selection when building trees and the final voting process for new samples is weighted. The weights for each classifier is affected by its weighted misclassification error.

The pseudocode for boosting is as given below:

Algorithm 1: The AdaBoost.M1

Input: Sequence of N examples, $S = \{(x_1, y_1), \ldots (x_N, y_N)\}$, $x_i \in X$, with labels $y_i \in Y = \{\omega_1, \ldots, \omega_C\}$, where ω_i is number of classes.
Weak classifier algorithm (*K*-Nearest Neighbor)
Number of Learning Rounds, T.

Initialize: Distribution: $D_i^t = \frac{1}{N}, i = 1, \ldots, N$
Neighbor(s): $K = m, 1 \le m \le N$

Do for: $t = 1, 2, \ldots, T$

1. Select the subset training data S_t drawn from the distribution D^t
2. Train the base classifier with S_t and receives the hypothesis h_t.
   $h_t : X \to Y$
3. Calculate the error of h_t.
   $h_t : \varepsilon_t = \sum_i^N D_i^t \times e_i^t$, where $e_i^t \begin{cases} (h_t(x_i) \neq y_i) = 1 \\ (h_t(x_i) = y_i) = 0 \end{cases}$
   If $\varepsilon_t > 0.5$, then set $T = t\text{-}1$ and abort the loop
4. Set weight, $\beta_t = \frac{\varepsilon_t}{1 - \varepsilon_t}$
5. Update distribution
   $D^t : D_i^{t+1} = \frac{D_i^t}{Z_t} \times \varphi_t$, where $\varphi_t \begin{cases} (h_t(x_i) = y_i) = \beta_t \\ (h_t(x_i) \neq y_i) = 1 \end{cases}$
   Where $Z_t = \sum_i^N D_i^t$ is normalization constant for D_i^{t+1} become a proper distribution.

Output: Given an unlabelled instance ***x***. Choose the class that have highest total vote as final classification.

$h_f(x) = \arg\max_{y \in Y} \sum_{t=1}^{T} \log \frac{1}{\beta_t} \times v$, where $v \begin{cases} (h_t(x) = \omega_j) = 1 \\ (h_t(x) \neq \omega_j) = 0 \end{cases}$

4.2 PROS AND CONS

Advantages:

- Higher accuracy
- Faster implementation

Disadvantages:

- Prone to overfitting problems
- Not as robust to errors and outliers

4.3 RESULTS OBTAINED

Performance measure obtained for 10 fold cross validation average scores, number of bins is set as 5, and number of classifiers is set as 10.

Results for project3 dataset1.txt are:

Accuracy: 98.24253075571178
Precision: 98.55769230769231
Recall: 96.69811320754716
F-Measure: 97.61904761904762

Results for project3 dataset2.txt are:

Accuracy: 76.62337662337663
Precision: 65.85365853658537
Recall: 67.5
F-Measure: 66.66666666666666

In the case of dataset1.txt, we can see that we achieve higher accuracy, which is the primary objective of boosting implementation. However, in dataset.txt results we see that there is a very high value for accuracy, which points towards a case of overfitting which is one of the major drawbacks of this algorithm. The best way to handle overfitting is by performing pruning on the decision tree during postprocessing.

5 NAÏVE BAYES

5.1 ALGORITHM DESCRIPTION

The Naive Bayes Classifier technique is based on the so-called Bayesian theorem and is particularly suited when the dimensionality of the inputs is high. Despite its simplicity, Naive Bayes can often outperform more sophisticated classification methods.

Bayes theorem works on conditional probability. Conditional probability is the probability that something will happen, *given that something else* has already occurred. Using the conditional probability, we can calculate the probability of an event using its prior knowledge.

$$P(H \mid E) = \frac{P(E \mid H) * P(H)}{P(E)}$$

- P(H) is the probability of hypothesis H being true. This is known as the prior probability.
- P(E) is the probability of the evidence(regardless of the hypothesis).
- P(E|H) is the probability of the evidence given that hypothesis is true.
- P(H|E) is the probability of the hypothesis given that the evidence is there. H is the hypothesis that a given data example X belong to particular class given the attribute values of X

Naive Bayes classifier assumes that all the features are **unrelated** to each other. Presence or absence of a feature does not influence the presence or absence of any other feature. In real datasets, we test a hypothesis given multiple evidence (feature). So, calculations become complicated.

To simplify the work, the feature independence approach is used to 'uncouple' multiple evidence and treat each as an independent one. Similar to decision trees, the best way to deal with continuous features in naïve bayes is to perform normalization of the data such that we are able to divide the data into specific categories.

Complexity of naive bayes is O(n) as only has to go through train set 1nce.

5.2 PROS AND CONS

Advantages:

- It can be easily train on small dataset.
- A simple algorithm depends on doing a bunch of counts.
- Great choice for Text Classification problems. It is a popular choice for spam email classification.

Disadvantages:

- It considers all the features to be unrelated, so it cannot learn the relationship between features.
- Basically Bayes can learn individual features importance but can't determine the relationship among features.

5.3 RESULTS OBTAINED

Results for project3 dataset1.txt are:

Accuracy: 89.47368421052632
Precision: 86.95652173913044
Recall: 86.95652173913044
F-Measure: 86.95652173913044

Results for project3 dataset2.txt are:

Accuracy: 68.08510638297872
Precision: 50.0
Recall: 53.333333333333336
F-Measure: 51.61290322580647

Solution for the problems with naive bayes dataset run:

- **For zero probability you can add noise so that probability does not become 0**
- **For continuous, use normal distribution model**.

Implementation:

- First convert the input dataset into a pandas dataframe then to array matrix
- Convert all elements in the array matrix to float values that can actually be converted
- Split the array matrix into training and test set, 90% training to 10% test
- Then we iterate over the training set to separate the rows containing 0 label from the rows containing 1 label and make a separate array for each of 1 and 0.(trainset_Zero, trainSet_One)
- Later we calculate the probability
- Method calculate probability of element in each column and create a map with the element as the key and its probability
- In the end we output p(H0|X) and p(H1|X), and whichever of this is higher, the label corresponding to that will be the label of the test row

6 CONCLUSION

From the results that we have obtained from the various algorithms, we are able to conclude that decision tree algorithm is the most accurate and efficient algorithm of the three for our datasets. In that, when we perform boosting with decision tree we see that this increases the efficiency and the performance measures as compared to the basic implementation of decision tree. We were able to learn the differences in the working of the classification algorithms outlined in this project and report.

We know that selecting a classification algorithm for any given dataset is a heuristic approach and the ideal choice will largely differ depending on the input that we are considering. However, by using experimentation, it is possible to identify the ideal algorithmic choice for any dataset.