SPL-1 Project Report

Classification of Data Using Decision Tree and Random Forest Based on Three Different Criteria

Submitted by

Pronob Karmoker

BSSE Roll No. : BSSE 1431

BSSE Session: 2021-2022

Submitted to

Dr.Mohammad Shoyaib

Professor

Institute of Information Technology

University of Dhaka

Supervisor's Approval: _____

(signature)

Institute of Information Technology

University of Dhaka

17-12-2023

Table of Contents

1. Introduction

This project is dedicated to the application of advanced classification methodologies, employing decision trees and random forests as the primary tools. The guiding principles for these methods include three specific measures: Entropy, Gini index, and Hellinger Distance. The central objective revolves around the core task of classification, leveraging the inherent structure provided by decision trees to make predictions based on input attributes. The project is structured around the following key components:

Splitting Criteria:

- Entropy Calculation: Determining optimal gain for effective decision-making by assessing uncertainty in the data.
- Gini Index: Evaluating impurity to guide the identification of optimal splits for classification.
- Hellinger Distance: Measuring the dissimilarity between probability distributions for effective decision boundaries.

Classification Metrics:

- Cross Validation: Ensuring model robustness through data partitioning for validation.
- Precision: Evaluating the accuracy of positive predictions made by the model.
- Recall: Assessing the model's ability to identify actual positive instances.
- Accuracy Calculation & F-Score: Calculating overall accuracy and striking a balance between precision and recall.

Decision Tree:

- Building three decision trees using the specified splitting criteria.
- Calculating gains based on these criteria and considering provided classification metrics to identify the optimal starting point for each decision tree.

Random Forest:

- Employing Random Forest, a collaborative ensemble of decision trees.
- Utilizing various splitting criteria to create a group of decision trees that work collectively to make predictions.
- Combining the results from individual trees to provide accurate and reliable outcomes.

Significance and Applications:

The project holds significant importance with applications across diverse domains. Key applications include:

- Medical Research: Facilitating the identification of risk factors for diseases by extracting representative subsets from large datasets.
- Finance: Enhancing the analysis of market trends and prediction of stock prices through the extraction of key data points.
- Marketing: Streamlining the analysis of consumer behavior and the development of effective marketing strategies.
- Image and Video Processing: Utilizing decision trees and random forests for classification tasks, contributing to more efficient image and video processing.

This project is poised to advance classification methods, offering valuable insights and predictions with enhanced accuracy and efficiency.

1.1 Background of the Project

➢**Entropy**: Entropy measures the level of disorder or uncertainty in a given dataset or system. It is a metric that quantifies the amount of information in a dataset, and it is commonly used to evaluate the quality of a model and its ability to make accurate predictions.

A higher entropy value indicates a more heterogeneous dataset with diverse classes, while a lower entropy signifies a more pure and homogeneous subset of data. Decision tree models can use entropy to determine the best splits to make informed decisions and build accurate predictive models.

$$Entropy\,(p) \;=\; -\sum_{i=1}^{N} p_i \log_2 p_i$$

➢ **Gini Index :** Gini Index or Gini impurity measures the degree or probability of a particular variable being wrongly classified when it is randomly chosen. the context of decision trees and random forests, the Gini index is employed as a measure of impurity or node impurity. When building a decision tree, the algorithm evaluates different features and splits the data based on the feature that minimizes the Gini index. The goal is to create splits that result in pure nodes, where all instances in a node belong to the same class.

$$Gini(node) = 1 - \sum_{i=1}^{n} p_i^2$$

➢**Hellinger Distance:** In the context of decision tree algorithms, the Hellinger Distance is used as a criterion for evaluating the impurity or node impurity when deciding how to split the data. The goal is to find splits that result in nodes with low impurity. The Hellinger Distance, along with other criteria like Gini index and entropy, helps in constructing decision trees that effectively classify instances into different classes.

$$H(P,Q) = \frac{1}{\sqrt{2}}\sqrt{\sum_i \left(\sqrt{p_i} - \sqrt{q_i}\right)^2}$$

Here, Pi and Qi are the probabilities associated with the *i*-th event in the sample space.

➢**Cross Validation:** Cross-validation is a statistical technique used in machine learning and model evaluation to assess the performance and generalizability of a predictive model. The primary purpose of cross-validation is to provide a more robust estimate of a model's performance by using different subsets of the data for training and testing.

The basic idea behind cross-validation is as follows:

1. **Data Splitting:** The original dataset is randomly divided into multiple subsets or folds. Common choices include k-fold cross-validation, where the data is split into k subsets, or stratified k-fold cross-validation, which ensures that each fold preserves the proportion of classes.

2. **Training and Testing:** The model is trained on some of the folds (training set) and then tested on the remaining fold (testing set). This process is repeated multiple times, with different folds used for testing in each iteration.

3. **Performance Metrics**: The model's performance is evaluated for each iteration, and the results are averaged or otherwise aggregated to obtain a more reliable estimate of how the model is likely to perform on unseen data.

The main advantage of cross-validation is that it helps to identify potential issues like overfitting or underfitting by providing a more realistic assessment of a model's performance across different subsets of the data. Common variations of cross-validation include k-fold cross-validation, leave-one-out cross-validation, and stratified cross-validation.

➢**Precision:** Precision is the ratio of true positive predictions to the total number of instances predicted as positive (the sum of true positives and false positives).Precision focuses on the accuracy of positive predictions. A higher precision indicates fewer false positives, meaning that the positive predictions made by the model are more reliable.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

➢**Recall:** Recall is the ratio of true positive predictions to the total number of actual positive instances (the sum of true positives and false negatives).Recall measures the model's ability to identify all relevant instances. A higher recall indicates fewer false negatives, meaning that the model successfully captures a larger proportion of actual positive instances.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

➢**F-Score:** The F1-score is the harmonic mean of precision and recall. It provides a balance between precision and recall, especially when there is an imbalance between the classes.The F1-score considers both false positives and false negatives. It is useful when there is an uneven class distribution, and you want a single metric that combines precision and recall.

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

➢**Decision tree:** In the context of classification, a decision tree is a supervised machine learning algorithm that is used to classify instances into predefined classes or categories. The decision tree makes decisions based on the values of input features, creating a tree-like structure where each internal node represents a decision based on a specific feature, and each leaf node represents a class label.

➢**Random Forest:** A Random Forest is an ensemble learning method in machine learning that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. In other words, it builds multiple decision trees and merges them together to get a more accurate and stable prediction.

1.2 Why Classification of Data Using Decision Tree and Random Forest

Classification of data using Decision Trees and Random Forests is a popular and effective approach due to several advantages these methods offer:

➢ **Interpretability:**

Decision Trees: Provide a transparent and easy-to-understand decision-making structure. Each node in the tree represents a decision based on a feature, allowing users to interpret and follow the logic of the model easily.

Random Forests:While not as interpretable as a single Decision Tree, Random Forests offer insights into feature importance, helping users understand which features contribute more to the model's predictions.

➢**Versatility:**

Decision Trees: Suitable for both classification and regression tasks. They can handle numerical and categorical data, making them versatile for various types of datasets.

Random Forests: An ensemble of Decision Trees, providing improved accuracy and robustness. They address the overfitting issues associated with individual Decision Trees and perform well across a range of applications.

➢**Accuracy:**

Decision Trees: Can capture complex relationships in the data, but may suffer from overfitting, especially with deep trees. Pruning techniques can help, but accuracy may still be limited.

Random Forests: By aggregating predictions from multiple trees, Random Forests enhance accuracy and generalize well to unseen data. They reduce overfitting and provide a more reliable prediction.

➢**Feature Importance:**

Decision Trees: Provide information on feature importance based on the structure of the tree.

Random Forests: Offer a more robust measure of feature importance by considering the average importance across multiple trees. This helps in identifying the most relevant features for classification.

➢**Reduced Risk of Overfitting:**

Decision Trees: Prone to overfitting, especially with deep trees that capture noise in the training data.

Random Forests: Mitigate overfitting by combining predictions from multiple trees, leading to a more generalized and reliable model.

2. Description of the Project

2.1 Read Data from file

```
int read_data()
{

    freopen(filename, "r", stdin);

    cin >> n >> d;

    lli i, j;

    for (i = 0; i < n; i++)
    {
        for (j = 0; j <= d; j++)
        {
            cin >> data[i][j];
        }
    }

    fclose(stdin);
}
```

The read_data() function serves as a crucial component in the project, responsible for reading and loading the dataset into the program.

2.2 Calculate Entropy

```
double entropy(double pos, double neg)
{
    if (pos == 0)
    {
        return 0;
    }
    else if (neg == 0)
    {
        return 0;
    }
    double store1 = -pos / (pos + neg) * log2(pos / (pos + neg));
    double store2 = neg / (pos + neg) * log2(neg / (pos + neg));
    double final = store1 - store2;
    return final;
}
```

2.3 Calculate Gini Index:

```
double gini_index(double pos, double neg)
{
    return 1 - ((pos / (pos + neg)) * (pos / (pos + neg)) + (neg / (pos + neg)) * (neg / (pos + neg)));
}
```

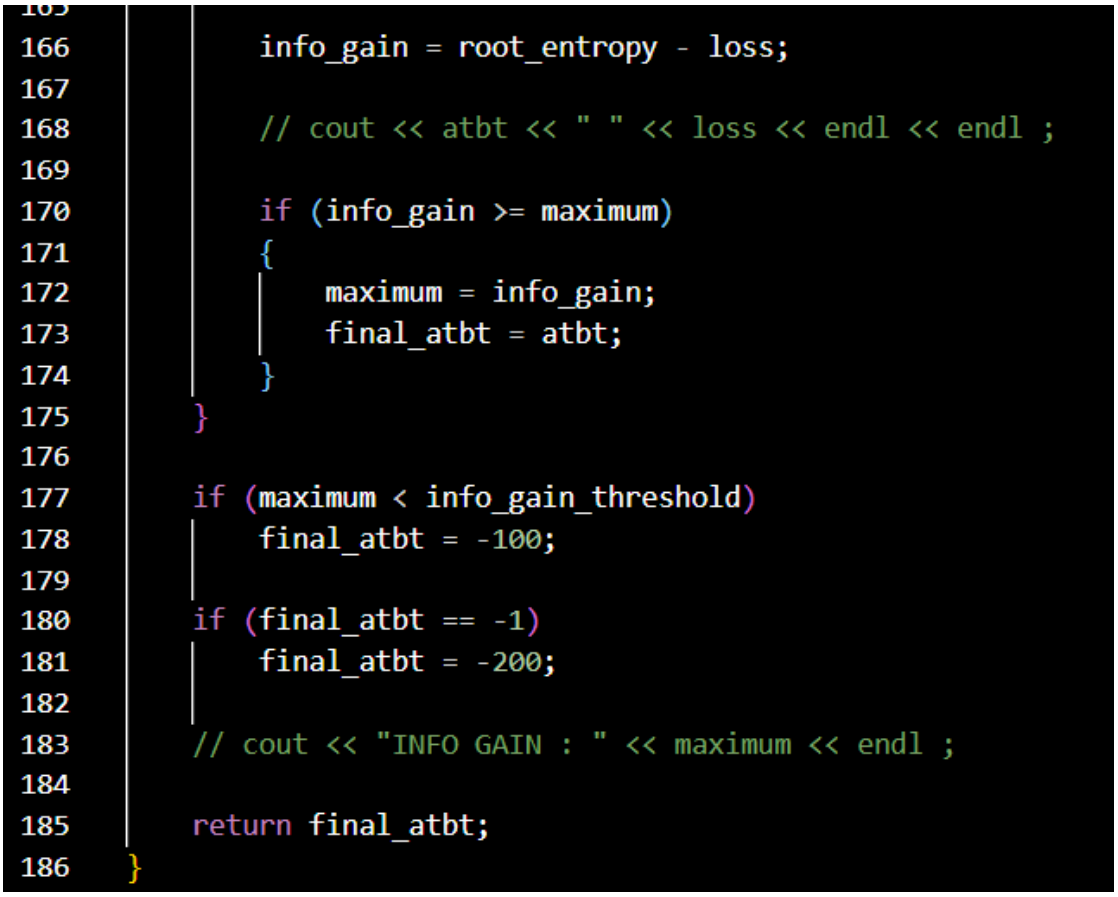
2.4 Calculate Hellinger Distance

```
double helinger_distance(double pos, double neg)
{
    return sqrt((pos / (pos + neg)) * (neg / (pos + neg)));
}
```

2.5 Calculate best attribute :

```
79   lli best_atbt(vector<int> a, double root_entropy)
80   {
81       double maximum = 0, info_gain, loss, temp;
82
83       lli final_atbt, atbt, total = a.size();
84
85       lli i, j;
86
87       map<pair<lli, lli>, lli> cfy; // classify
88
89       final_atbt = -1;
90
91       for (atbt = 0; atbt < d; atbt++)
92       {
93           if (atbt_list[atbt] > 0)
94               continue;
95
96           cfy.clear();
97
98           loss = 0;
99
100          for (i = 0; i < total; i++)
101          {
102              cfy[make_pair(data[a[i]][atbt], data[a[i]][d])]++;
103          }
104
105          map<pair<lli, lli>, lli>::iterator it, next;
106
```

```
107         for (it = cfy.begin(); it != cfy.end();)
108         {
109
110             double entropy, pplus, pminus, pall;
111
112             next = ++it;
113             it--;
114
115             if (next == cfy.end())
116             {
117                 //  cout << it->first.first << " " << it->first.second << endl ;
118
119                 entropy = 0;
120                 pall = it->second;
121                 it++;
122             }
123             else
124             {
125                 if (it->first.first == next->first.first)
126                 {
127                     //  cout << it->first.first << " " << it->first.second << endl ;
128                     //  cout << next->first.first << " " << next->first.second << endl ;
129
130                     entropy = 0;
131
132                     pminus = it->second;
133                     pplus = (next)->second;
135                     pall = pplus + pminus;
136
137                     pplus = pplus / pall;
138                     pminus = pminus / pall;
139
140                     entropy = -(pplus * log2(pplus) + pminus * log2(pminus));
141
142                     it++;
143                     it++;
144                 }
145                 else
146                 {
147                     // cout << it->first.first << " " << it->first.second << endl ;
148
149                     entropy = 0;
150                     pall = it->second;
151                     it++;
152                 }
153             }
154
155             // cout << endl ;
156
157             temp = pall / (double)total;
158
159             // cout << pall << " " << total << endl ;
160             // cout << temp << endl << endl  ;
161
162             temp = temp * entropy;
163             loss = loss + temp;
164         }
```



```
166             info_gain = root_entropy - loss;
167
168             // cout << atbt << " " << loss << endl << endl ;
169
170             if (info_gain >= maximum)
171             {
172                 maximum = info_gain;
173                 final_atbt = atbt;
174             }
175         }
176
177         if (maximum < info_gain_threshold)
178             final_atbt = -100;
179
180         if (final_atbt == -1)
181             final_atbt = -200;
182
183         // cout << "INFO GAIN : " << maximum << endl ;
184
185         return final_atbt;
186     }
```

By calculating information gain I find the best attribute .

2.6 Recursive Node Splitting and Decision Tree Construction

```
239  int split(node *curr, int depth)
240  {
241
242      double root_entropy, pplus, pminus;
243
244      if (max_depth < depth)
245          max_depth = depth;
246
247      lli i, total = curr->set.size();
248
249      for (i = 0; i < total; i++)
250      {
251          if (data[curr->set[i]][d] == 1)
252              curr->plus++;
253          else
254              curr->minus++;
255      }
256
257      // Calculating entropy
258
259      pplus = (double)curr->plus / (double)total;
260      pminus = (double)curr->minus / (double)total;
261
262      root_entropy = -(pplus * log2(pplus)) - (pminus * log2(pminus));
263
264      // Done
```

```
268      double purity;
269
270      purity = curr->plus;
271      purity = purity / (double)total;
272
273      if (purity >= early)
274      {
275          curr->label = 1;
276
277          // cout << "Added plus : All pure values\n\n" ;
278
279          return 0;
280      }
281
282      purity = curr->minus;
283      purity = purity / (double)total;
284
285      if (purity >= early)
286      {
287          curr->label = 0;
288
289          // cout << "Added minus : All pure values\n\n" ;
290
291          return 0;
292      }
```

```
296     lli atbt = best_atbt(curr->set, root_entropy); //  Returns best
297
298     if (atbt == -100) //
299     {
300         curr->label = data[curr->set[0]][d];
301
302         // cout << "Added label : Zero Info Gain\n\n" ;
303
304         return 0;
305     }
306
307     if (atbt == -200)
308     {
309         if (curr->plus > curr->minus)
310             curr->label = 1;
311         else
312             curr->label = 0;
313
314         // cout << "Added label : All attributes over\n\n" ;
315
316         return 0;
317     }
```

```
319     curr->atbt = atbt;
320
321     atbt_list[atbt]++; // You used up that attribute !!!
322
323     for (i = 0; i < total; i++)
324     {
325         lli element = curr->set[i];
326         lli key = data[element][atbt];
327
328         if (curr->child[key] == NULL)
329         {
330             curr->child[key] = new node();
331
332             curr->child[key]->current = key;
333
334             curr->child[key]->parent = curr;
335
336             curr->child[key]->set.push_back(element);
337         }
338         else
339         {
340             curr->child[key]->set.push_back(element);
341         }
342     }
```

```
365     map<lli, node *>::iterator it;
366
367     for (it = curr->child.begin(); it != curr->child.end(); it++)
368     {
369         split(it->second, depth + 1);
370     }
371
372     atbt_list[atbt]--; // You freed that attribute !!!
373
374     return 0;
375 }
```

The split function is the core of decision tree construction. It manages the splitting of nodes based on selected attributes, handles early stopping conditions, calculates entropy, and recursively constructs the decision tree structure. This function is a fundamental part of training decision tree models in machine learning.

2.7 Recursive Decision Tree Evaluation

```
377 int evaluate(int index, node *curr)
378 {
379
380     if (curr->label != -1)
381     {
382         return curr->label;
383     }
384
385     lli atbt = curr->atbt;
386
387     if (curr->child[data[index][atbt]] != NULL)
388     {
389         return evaluate(index, curr->child[data[index][atbt]]);
390     }
391     else
392     {
393         if (curr->plus > curr->minus)
394             return 1;
395         else
396             return 0;
397     }
398 }
```

this function is crucial for traversing a decision tree to determine the classification label for a given input data point. It recursively navigates the tree based on the attribute values of the input data until a leaf node (a node with a label) is reached. This label is then returned as the output of the decision tree for that specific input.

2.8 Extract Non Null Child Nodes

```
400  vector<node *> getChild(map<lli, node *> m)
401  {
402
403      vector<node *> ch;
404
405      map<lli, node *>::iterator it;
406
407      for (it = m.begin(); it != m.end(); it++)
408      {
409          if ((it->second) != NULL)
410              ch.push_back(it->second);
411      }
412
413      return ch;
414
415  } // end getChild
```

The purpose of this function is to extract non-NULL child nodes from a map and return them in the form of a vector.

2.9 Count Nodes And Attributes

This function performs a breadth-first traversal of a tree, counting the nodes and updating counts associated with the attributes of each node. The attributes and counts are stored in the attrCount data structure.

```
417  lli countNodes(node *root)
418  {
419      lli c = 0, i;
420
421      deque<node *> q;
422
423      if (root == NULL)
424          return 0;
425
426      q.push_front(root);
427
428      while (!q.empty())
429      {
430          node *rt = q.front();
431
432          attrCount[rt->atbt]++;
433
434          q.pop_front();
435
436          c++;
437
438          vector<node *> nb = getChild(rt->child);
439
440          for (i = 0; i < nb.size(); i++)
441          {
442              if ((nb[i]) != NULL)
443                  q.push_front(nb[i]);
444          }
445      } // end while
446
447      return c;
448  } // end countNOdes
```

2.10 Add Noise To Data

This function introduces noise to the dataset by randomly flipping the class labels (0 or 1) of a specified percentage of data points. It takes a parameter p representing the percentage of data points to be affected by noise. The noise is introduced by randomly selecting data points and changing the class label (0 to 1 or 1 to 0). The randomness is achieved using the rand() function with a seed based on the current time.

```
450  void addNoise(double p)
451  {
452      p /= 100;
453      srand(time(NULL));
454      // cout<<n*p;
455      for (int i = 0; i < n * p; i++)
456      {
457
458          int a = rand() % n; // ,b=rand()%n;
459          int temp = data[a][d];
460          data[a][d] = 1 - data[a][d]; // data[b][d];
461          // data[b][d]=1-data[b][d];//;t
462          // cout<<"swap "<<a<<", "<<b<<"
463      }
464  }
```

2.11 Initialize Root Node With Subset

```
494  int init_head(node *head)
495  {
496      lli i;
497
498      max_depth = 0;
499
500      for (i = 0; i < splindex; i++)
501      {
502          head->set.push_back(test_split[i]);
503      }
504  }
```

The init_head function appears to initialize the root node of a decision tree with a subset of the dataset designated for training.

2.12 Calculate Classification Matric

2.12.1 precision , recall , F1 score

```cpp
tuple<double, double, double> evaluate(const vector<int> &trueLabels, const vector<int> &predictedLabels)
{
    int truePositive = 0, falsePositive = 0, falseNegative = 0;

    for (size_t i = 0; i < trueLabels.size(); ++i)
    {
        if (trueLabels[i] == 1 && predictedLabels[i] == 1)
        {
            truePositive++;
        }
        else if (trueLabels[i] == 0 && predictedLabels[i] == 1)
        {
            falsePositive++;
        }
        else if (trueLabels[i] == 1 && predictedLabels[i] == 0)
        {
            falseNegative++;
        }
    }

    double precision = (truePositive + falsePositive > 0) ? static_cast<double>(truePositive) / (truePositive + falsePositive) : 0.0;
    double recall = (truePositive + falseNegative > 0) ? static_cast<double>(truePositive) / (truePositive + falseNegative) : 0.0;
    double f1 = (precision + recall > 0) ? (2 * precision * recall) / (precision + recall) : 0.0;

    return make_tuple(precision, recall, f1);
}
```

2.12.2 K fold Cross Validation

```cpp
// 10-fold cross-validation
void kFoldCrossValidation(const vector<vector<double>> &features, const vector<int> &labels, int k = 10)
{
    size_t dataSize = features.size();
    size_t foldSize = dataSize / k;

    for (int fold = 0; fold < k; ++fold)
    {
        // Split data into training and testing sets
        vector<vector<double>> trainingFeatures, testingFeatures;
        vector<int> trainingLabels, testingLabels;

        for (size_t i = 0; i < dataSize; ++i)
        {
            if (i >= fold * foldSize && i < (fold + 1) * foldSize)
            {
                testingFeatures.push_back(features[i]);
                testingLabels.push_back(labels[i]);
            }
            else
            {
                trainingFeatures.push_back(features[i]);
                trainingLabels.push_back(labels[i]);
            }
        }
```

```
570
571         // Train your model on training data
572         trainModel(trainingFeatures, trainingLabels);
573
574         // Make predictions on the testing data
575         vector<int> predictions = predict(testingFeatures);
576
577         // Evaluate the model
578         auto [precision, recall, f1] = evaluate(testingLabels, predictions);
579
580         // Print or store the evaluation metrics
581         cout << "Fold " << fold + 1 << " - Precision: " << precision << ", Recall: " << recall << ", F1 Score: " << f1 << endl;
582     }
583 }
```

2.13 Evaluate Decision Tree Performance

```
585 double run()
586 {
587     lli result = 0, i;
588
589     double error = 0;
590
591     for (i = splindex; i < n; i++)
592     {
593         result = evaluate(test_split[i], head);
594
595         if (result != data[test_split[i]][d])
596             error++;
597     }
598
599     error = error / (double)(n - splindex);
600
601     // cout << "Depth : " << max_depth << endl << "Error :
602
603     return error;
604 }
605
```

this function evaluates the performance of a decision tree on a set of test examples and returns the error rate.

2.14 Random Forest :

```
668  int ensemble(lli num)
669  {
670      node *forest[num];
671
672      lli i, j;
673
674      vector<lli> atbt_shuffle;
675
676      for (i = 0; i < d; i++)
677      {
678          atbt_shuffle.push_back(i);
679      }
680
681      lli k;
682
683      lli plus_count, minus_count;
684
685      for (k = 0; k < num; k++)
686      {
687          atbt_list.clear();
688
689          forest[k] = new node();
690          forest[k]->parent = NULL;
691
692          random_shuffle(atbt_shuffle.begin(), atbt_shuffle.end());
693
694          for (i = 9; i < d; i++)
695          {
696              atbt_list[atbt_shuffle[i]] = 100;
697          }
698
699          init_head(forest[k]);
700
701          split(forest[k], 0);
702
703          // cout << "Depth : " << max_depth << " Nodes : " << co
704      }
705
706      lli result = -1;
707
708      double test_error = 0, train_error = 0;
709
710      for (i = splindex; i < n; i++)
711      {
712          plus_count = minus_count = 0;
713
714          for (k = 0; k < num; k++)
715          {
716              result = evaluate(test_split[i], forest[k]);
717
718              if (result == 1)
719                  plus_count++;
720              else
721                  minus_count++;
722          }
723
724          if (plus_count > minus_count)
725              result = 1;
726          else
727              result = 0;
728
729          if (result != data[test_split[i]][d])
730              test_error++;
731      }
```

```
732
733     for (i = 0; i < splindex; i++)
734     {
735         plus_count = minus_count = 0;
736
737         for (k = 0; k < num; k++)
738         {
739             result = evaluate(test_split[i], forest[k]);
740
741             if (result == 1)
742                 plus_count++;
743             else
744                 minus_count++;
745         }
746
747         if (plus_count > minus_count)
748             result = 1;
749         else
750             result = 0;
751
752         if (result != data[test_split[i]][d])
753             train_error++;
754     }
755
756     test_error = test_error / (double)(n - splindex);
757     test_error = 100 * (1 - test_error);
758
759     train_error = train_error / (double)splindex;
760     train_error = 100 * (1 - train_error);
761
762     cout << setw(20) << train_error << setw(20) << test_error << endl;
763 }
```

In this function, num represents the number of trees in the random forest. The function creates an ensemble of decision trees (forest) by repeatedly calling the split function for each tree. The predictions of each tree in the ensemble are then combined to make the final prediction. The training and test accuracies of the random forest for different numbers of trees are printed.

3. User Interface

We build decision trees and random forests for a insurance dataset, evaluating it for various experiments such as adding noise and tree pruning. Dataset taken from :

[Insurance Company Benchmark (COIL 2000)]

3.1 HOW TO RUN :

➢Compile the program by entering the following command

g++ -o ID3 ID3.cpp

➢Run the executable by entering the following command

./ID3 ticdata2000.txt experiment_no

Our project will start from the main function.

```
765  int main(int argc, char *argv[])
766  {
767
768      filename = argv[1];
769
770      exptno = argv[2];
771
772      initialise();
773
774      lli i, j;
775
776      if (exptno[0] == '1')
777      {
778          cout << setw(30) << "Experiment no. - 1\n\n";
779
780          double iter;
781
782          int runs = 10, nnodes = 0;
783
784          cout << "Threshold %" << setw(16) << "Test accuracy" << setw(16) << "No. of nodes" << endl
785              << endl;
786
787          for (iter = 0.94; iter <= 1; iter = iter + 0.01)
788          {
789
790              for (int i = 0; i <= d; i++)
791              {
792                  attrCount[i] = 0;
793              }
794
795              double res[3] = {0.0};
```

```
795         double res[3] = {0.0};
796
797         for (int rr = 0; rr < runs; rr++)
798         {
799
800             early = iter;
801
802             head = new node();
803             head->parent = NULL;
804
805             split_data(1000);
806
807             init_head(head);
808
809             split(head, 0);
810
811             nnodes = countNodes(head);
812
813             if (nnodes == 1)
814             {
815                 rr--;
816                 continue;
817             }
818
819             res[0] += run();
820             res[1] += nnodes;
821         }
822
823         res[0] /= runs;
824         res[1] /= runs;
825
826         cout << setw(4) << (iter * 100) << setw(20) << (100 - res[0] * 100) << setw(16) << (int)res[1] << endl;
827     }
```

```
829         cout << "\nNumber of times an attribute is used as the splitting function : " << endl
830             << endl;
831
832         vector<pair<int, int>> afreq;
833
834         for (i = 0; i <= d; i++)
835         {
836             attrCount[i] /= runs;
837             afreq.push_back(make_pair(attrCount[i], i));
838         }
839
840         sort(afreq.begin(), afreq.end());
841
842         for (i = afreq.size() - 1; i >= 0; i--)
843         {
844             if (afreq[i].first > 0)
845                 cout << setw(4) << afreq[i].second << " -> " << afreq[i].first << endl;
846         }
847     }
848     else if (exptno[0] == '2')
849     {
850         cout << setw(30) << "Experiment no. - 2\n\n";
851
852         int iter = 4;
853
854         double noise[] = {0.5, 1, 5, 10};
855
856         int runs = 10, nnodes = 0;
857
858         cout << "Noise %" << setw(16) << "Test accuracy" << setw(16) << "No. of nodes" << endl
859             << endl;
860
```

```
861         for (iter = 0; iter < 4; iter++)
862         {
863
864             read_data();
865             addNoise(noise[iter]);
866
867             double res[3] = {0.0};
868
869             for (int rr = 0; rr < runs; rr++)
870             {
871
872                 head = new node();
873                 head->parent = NULL;
874
875                 split_data(1000);
876
877                 init_head(head);
878
879                 split(head, 0);
880
881                 nnodes = countNodes(head);
882
```

```
882
883                 if (nnodes == 1)
884                 {
885                     rr--;
886                     continue;
887                 }
888
889                 res[0] += run();
890                 res[1] += nnodes;
891             }
892
893             res[0] /= runs;
894             res[1] /= runs;
895
896             cout << setw(4) << noise[iter] << setw(16) << (100 - res[0] * 100) << setw(16) << (int)res[1] << endl;
897         }
898     }
```

```
899     else if (exptno[0] == '3')
900     {
901         cout << setw(24) << "Experiment no. - 3\n\n";
902
903         cout << "Test accuracy" << setw(16) << "No. of nodes" << endl
904             << endl;
905
906         head = new node();
907         head->parent = NULL;
908
909         split_data(1000);
910
911         init_head(head);
912
913         split(head, 0);
914
915         global_error = pruntest();
916
917         pruning(head);
918
919         cout << endl;
920     }
```

```
921     else if (exptno[0] == '4')
922     {
923         cout << setw(34) << "Experiment no. - 4\n\n";
924
925         cout << "No. of trees" << setw(16) << "   Training accuracy" << setw(16) << "Test accuracy" << endl
926             << endl;
927
928         for (i = 1; i <= 25; i++)
929         {
930             split_data(1000);
931             cout << setw(4) << i;
932             ensemble(i);
933         }
934     }
935 }
936
```

Output :

```
PS C:\Users\Shuvo\OneDrive\Desktop\SPL-1\Final>    ./ID3  ticdata2000.txt 1
          Experiment no. - 1

Threshold %   Test accuracy    No. of nodes

  94              90.2509             229
  95              91.4849             215
  96              91.3501             229
  97              91.0908             242
  98              91.1945             238
  99              90.7321             271
 100              90.3256             287

Number of times an attribute is used as the splitting function :

  79 -> 8
  67 -> 6
  58 -> 4
  84 -> 2
  75 -> 2
  64 -> 2
  41 -> 2
  38 -> 2
  37 -> 2
  82 -> 1
  81 -> 1
  69 -> 1
  46 -> 1
  40 -> 1
  36 -> 1
  35 -> 1
  30 -> 1
  27 -> 1
  22 -> 1
  16 -> 1
   0 -> 1
```

```
PS C:\Users\Shuvo\OneDrive\Desktop\SPL-1\Final>    ./ID3  ticdata2000.txt 2
          Experiment no. - 2

Noise %   Test accuracy    No. of nodes

 0.5          89.9564             298
   1          89.6682             292
   5          84.5852             386
  10          80.1389             458
```

```
PS C:\Users\Shuvo\OneDrive\Desktop\SPL-1\Final>    ./ID3  ticdata2000.txt 3
    Experiment no. - 3

Test accuracy    No. of nodes

 90.6885             237
 90.7715             236
 90.9166             235
 90.9996             234
 91.0411             233
 91.3107             232
 91.4558             231
 91.5595             230
 91.7047             229
 91.7669             228
 91.9328             227
 92.1817             224
 92.2854             223
 92.3891             222
 92.4098             221
 92.6586             220
 92.7623             219
```

```
PS C:\Users\Shuvo\OneDrive\Desktop\SPL-1\Final>    ./ID3  ticdata2000.txt 4
            Experiment no. - 4

No. of trees   Training accuracy   Test accuracy

   1               95.6              89.3613
   2               95.9              93.7163
   3               96.1              93.1771
   4               98.9              93.4052
   5               98.1              93.5297
   6               97.9              92.0987
   7               97.6              93.9859
   8               97.1               93.426
   9               96.9               92.949
  10               96.7              94.1518
  11               97.5              93.6126
  12                 97              92.5757
  13               97.4              92.4305
  14               95.8              93.3015
  15               97.7              93.7785
  16               97.8              93.6748
  17               96.7              93.4882
  18               97.9              94.0066
  19               97.6              92.8246
  20               97.6              93.8822
  21                 98              93.6748
  22               97.3              93.1149
  23               97.4              92.7623
  24               95.7              93.1564
  25                 98              93.3637
```

3.2 Experiments :

1. We vary the "stopping criteria" that prevents further splitting of node. Changes in accuracy and complexity of model are observed.
2. Add noise to the dataset and evaluate the accuracy of the model along with the change in its complexity (number of nodes)
3. Perform "Reduced Error Pruning" on the tree and measure the change in accuracy of the tree.
4. Create a random forest where we select a subset of features, make multiple trees, and take majority vote for the result.

4. Challenges Faced

➢ **Implementation of the Algorithms :** There was a little bit difficulty to implement the ID3 algorithm.

➢ **Manage Large Codes** : Working with a large codebase is a tedious task. It becomes very difficult to track changes on different files.

➢ **Debugging codes** : Frequently, I got segmentation problems or dumped code errors but I was not sure where it happening. Then this issue was instantly solved by debugging the CPP files.

➢**Finding the practical Dataset**

5. Conclusion

I learned:

- ❖ Decision Tree Algorithms
- ❖ Implementation Skills
- ❖ Tree Construction
- ❖ Testing and Evaluation
- ❖ Optimization and Early Stopping
- ❖ Experimental Design
- ❖ Data Manipulation
- ❖ Visualization

Future Project Extensions:

- ❖ Handling Imbalanced Data
- ❖ Integration with Other Models
- ❖ User Interface
- ❖ Real-world Application
- ❖ Continuous Learning

Reference [new page]

[1] Insurance Company Benchmark (COIL 2000) 12/12/2023

[2] https://ijaers.com/uploads/issue_files/60IJAERS-04202057-Iterative.pdf 13/12/2023

[3] https://www.researchgate.net/publication/259235118_Random_Forests_and_Decision_Trees Jehad Ali1 , Rehanullah Khan2 , Nasir Ahmad3 , Imran Maqsood4 13/12/2023

[4] https://en.wikipedia.org/wiki/Hellinger_distance 16/12/2023