Random Forest
Course Project 1

Prepared for:

CS529

Prepared by:

Team
Kaneez Fatima and Muhammad

Report on Random Forest Project

Summary: The project goal is to make a Random Forest predictive algorithm to determine credit card fraud. The training and test data were provided through Kaggle. The data included several features that include categorical and numerical data. The algorithm achieved an overall maximum accuracy of 72.448%.

Answer to Rubric Questions:

1. What is the difference between the resulting trees that each IG method produced? Compare and contrast in terms of accuracy and structure.

   Answer:

- **Entropy** : Produces slightly deeper trees on average, because it is *more sensitive* to changes in low-probability (minority) classes. It prefers splits that give "purer" nodes even for small class gains.
- **Gini Index**: Produces more compact trees (slightly fewer levels) because it emphasizes overall class purity but is less sensitive to small minority improvements.
- **Misclassification Error**: Produces very shallow trees, since it only measures whether the majority label changes and ignores finer class proportions.

Structure and Accuracy comparison:

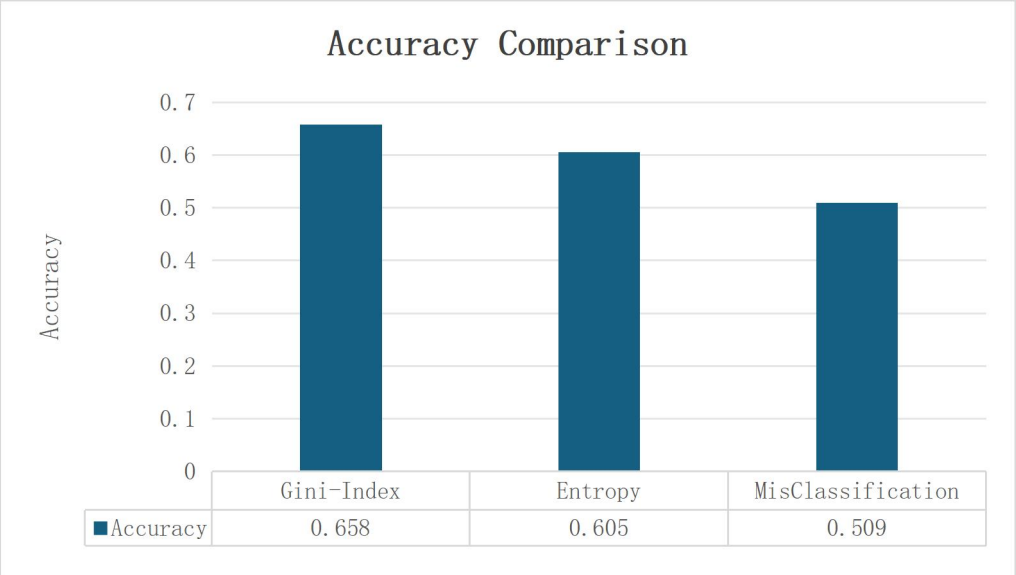


Fig. 1: A bar plot showing effect of IG methods on the accuracy of the trees

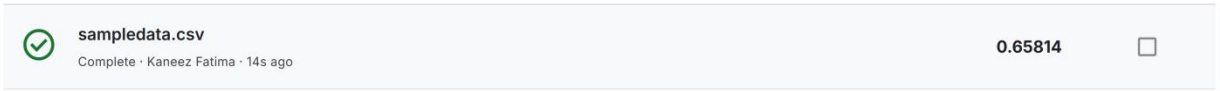
Some representative screenshots showing accuracy of the trees corresponding to structures using different IG methods are shown below:

Accuracy of a representative tree with IG as Misclassification Error

Structure of a representative tree using Misclassification Error as IG

```
    Predict → 0.0
  If card2 == 162:
    Predict → 0.0
  If card2 == 157:
    Predict → 0.0
  If card2 == 113:
    [Split on: C6]
      If C6 == (1.0, 2.0]:
        [Split on: C13]
          If C13 == (2.0, 5.0]:
            Predict → 0.0
          If C13 == (17.0, 2918.0]:
            Predict → 0.0
          If C13 == (1.0, 2.0]:
            Predict → 0.0
          If C13 == (5.0, 17.0]:
            Predict → 0.0
          If C13 == (-0.001, 1.0]:
            Predict → 1.0
      If C6 == (2.0, 2252.0]:
        Predict → 0.0
      If C6 == (-0.001, 1.0]:
        Predict → 0.0
  If card2 == 568:
    Predict → 0.0
  If card2 == 593:
    Predict → 0.0
  If card2 == 381:
    Predict → 0.0
  If card2 == 249:
    [Split on: C6]
      If C6 == (-0.001, 1.0]:
        Predict → 0.0
      If C6 == (1.0, 2.0]:
        Predict → 0.0
      If C6 == (2.0, 2252.0]:
        [Split on: C13]
          If C13 == (5.0, 17.0]:
            Predict → 1.0
          If C13 == (17.0, 2918.0]:
            Predict → 0.0
```

Accuracy of a representative tree with IG as as Gini-index

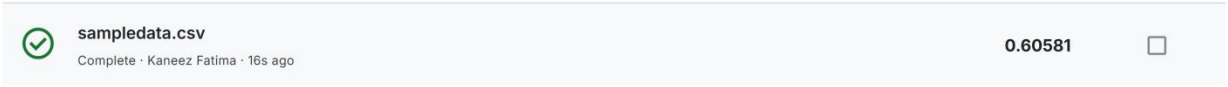


Structure of a representative tree using as Gini-index

```
Tree 48:
[Split on: C9]
  If C9 == (-0.001, 1.0]:
    [Split on: card6]
      If card6 == debit:
        Predict → 0.0
      If card6 == credit:
        [Split on: C6]
          If C6 == (2.0, 2252.0]:
            [Split on: C8]
              If C8 == (1.0, 3332.0]:
                [Split on: C10]
                  If C10 == (1.0, 3256.0]:
                    [Split on: card3]
                      If card3 == (150.0, 231.0]:
                        Predict → 1.0
                      If card3 == (99.999, 150.0]:
                        Predict → 1.0
                  If C10 == (-0.001, 1.0]:
                    Predict → 1.0
              If C8 == (-0.001, 1.0]:
                [Split on: card3]
                  If card3 == (99.999, 150.0]:
                    Predict → 1.0
                  If card3 == (150.0, 231.0]:
                    Predict → 1.0
          If C6 == (-0.001, 1.0]:
            Predict → 1.0
          If C6 == (1.0, 2.0]:
            [Split on: C8]
              If C8 == (-0.001, 1.0]:
                Predict → 1.0
              If C8 == (1.0, 3332.0]:
                [Split on: card3]
                  If card3 == (99.999, 150.0]:
                    Predict → 1.0
                  If card3 == (150.0, 231.0]:
                    Predict → 1.0
      If card6 == nan:
        Predict → debit
```

```
        If card1 == 1151.0:
          Predict → 0.0
    If C10 == (1.0, 3256.0]:
      [Split on: card1]
        If card1 == 16746.0:
          Predict → 0.0
        If card1 == 5938.0:
          Predict → 1.0
        If card1 == 9002.0:
          [Split on: C13]
            If C13 == (17.0, 2918.0]:
              Predict → 1.0
            If C13 == (2.0, 5.0]:
              Predict → 0.0
        If card1 == 10876.0:
          Predict → 1.0
        If card1 == 16136.0:
          Predict → 0.0
        If card1 == 16314.0:
          Predict → 0.0
        If card1 == 12839.0:
          [Split on: C13]
            If C13 == (-0.001, 1.0]:
              Predict → 1.0
            If C13 == (1.0, 2.0]:
              Predict → 0.0
        If card1 == 1047.0:
          Predict → 1.0
        If card1 == 9633.0:
          Predict → 1.0
        If card1 == 12511.0:
          Predict → 1.0
        If card1 == 10175.0:
          Predict → 1.0
        If card1 == 5365.0:
          Predict → 1.0
```

Accuracy of a representative tree with IG as Entropy



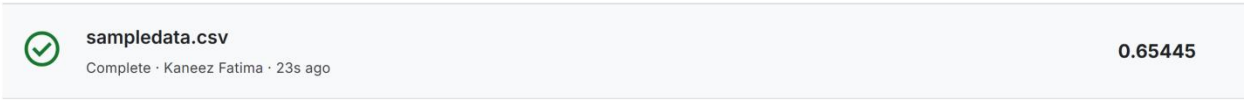
Structure of a representative tree using Entropy

```
    Predict → 1.0
  If card1 == 14321.0:
    Predict → 0.0
  If card1 == 2587.0:
    Predict → 0.0
  If card1 == 1479.0:
    Predict → 0.0
  If card1 == 15718.0:
    Predict → 0.0
  If card1 == 9136.0:
    Predict → 0.0
  If card1 == 13416.0:
    Predict → 0.0
  If card1 == 3111.0:
    Predict → 1.0
  If card1 == 1797.0:
    Predict → 0.0
  If card1 == 14511.0:
    Predict → 0.0
  If card1 == 9965.0:
    Predict → 0.0
  If card1 == 17904.0:
    Predict → 0.0
  If card1 == 4270.0:
    Predict → 0.0
  If card1 == 4100.0:
    Predict → 0.0
  If card1 == 17247.0:
    Predict → 0.0
  If card1 == 8924.0:
    Predict → 0.0
  If card1 == 4053.0:
    Predict → 0.0
  If card1 == 4097.0:
    Predict → 0.0
  If card1 == 12961.0:
    Predict → 0.0
  If card1 == 16363.0:
```

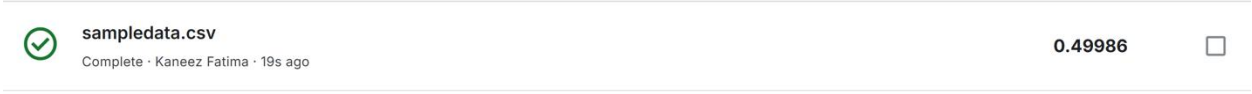
1. How did the value of alpha affect Chi Square termination? Compare and contrast in terms of accuracy and structure.

Answer: With alpha values very low such as 0.01 requires stronger significance to justify a split in tree construction. It also makes stricter condition to termination resulting in fewer splits and smaller trees but with wider structure. It leads to moderate accuracy since it does not allow take enough details from the data and maybe more impurity may present while predicting the right class. In contrast, with high alpha values such as 0.1, 0.25 and more, the splitting criteria are easy to achieve resulting in frequent splits and larger and complex trees in length. Since high alpha allows more splits, it investigates more feature data than it may require and likely to cause overfitting but higher accuracy was seen until 0.25. The accuracy is improved at the expense of computation cost. Below are some screenshots from Kaggle for accuracy numbers corresponding to α values for Chi-Square test.

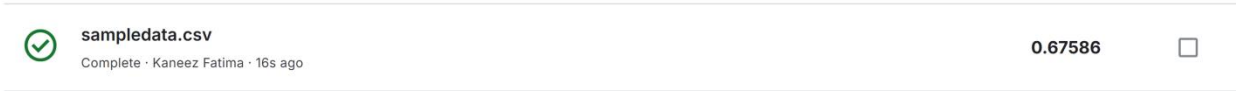
Accuracy with $\alpha = 0.001$



Accuracy with $\alpha = 0.05$



Accuracy with $\alpha = 0.01$



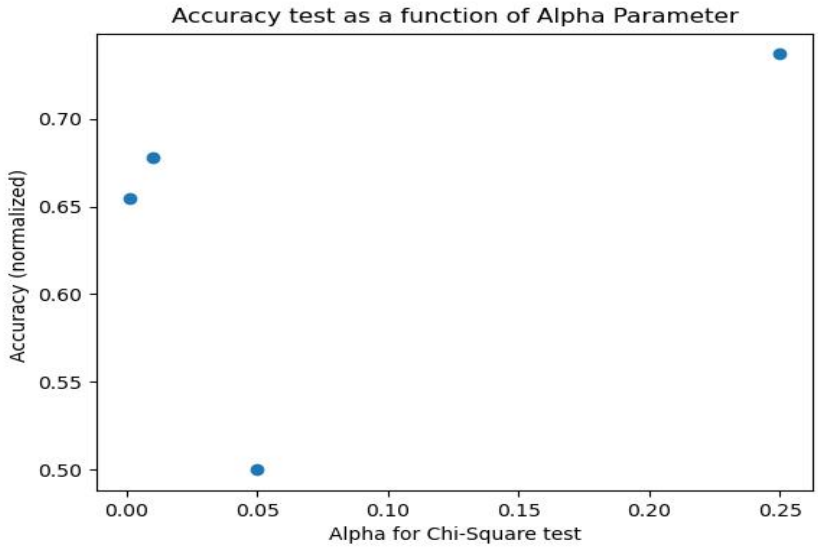


Fig. 2: A scatter plot showing effect of parameter, Alpha(α) on the accuracy of the trees

2. Which Information Gain Criteria method did you choose for your Final (kaggle submission) Random Forest? Why?

Answer:

- The giniIndex() is computationally cheaper as it does not contain logarithms.
- For the balanced bootstrap sampling and class weights, Gini and Entropy yield almost identical splits, so we save computation without loss of performance.
- The accuracy achieved from giniIndex() is relatively higher than other impurity methods

3. What did you do to account for missing data?

Answer:

- In processNumericData(), we replaced "NotFound" with NaN, then filled missing values using the **median** of each numeric feature.
- For categorical features, we treated missing categories implicitly by not dropping rows and leaving them as "NotFound" to consider them as a separate branch letter in the tree splitting.
- we avoided dropping rows, preserving rare fraud instances.

4. What did you do to account for numerical features?

Answer:

- Converted NotFound columns to nan then Converted numeric columns with pd.to_numeric() to impute the missing data with median values.
- Applied quantile binning (qcut) when:
    - Feature skew > 1.0, or
    - A single value dominated (>50% frequency of the single attribute).
- Quantile binning discretizes heavily skewed features into equal-frequency bins (provide ranges), which eventually reduces noise.
- This helps the tree splits be more balanced (not all thresholds fall in the majority region).

5. What did you do to account for class imbalance?

Answer: We addressed imbalance in two ways:

1. Balanced bootstrap sampling (balancedBootstrap function)
    - Each tree gets equal samples of majority and minority classes.
    - This ensures every tree learns from fraud cases.
2. Class weights (computeClassWeights function)
    - Multiplied the probabilities with class weights/frequencies in each impurity method
    - This affects Gini/Entropy calculations directly and Weighted impurities prevent splits that ignore minority classes.

6. What other optimizations did you implement to improve either accuracy or performance?

Answer: There are several implementation methods we chose to improve both accuracy and performance.

i) Maximum Depth as hyper parameter: We used Gini-index to create shorter trees and tried to avoid overfitting. In addition, we used Maximum depth as hyperparameter in our random forest algorithm. Using Gini-index helped to run the code faster for better performance and setting a Maximum depth allowed to trim the trees shorter to again avoiding overfitting. We found maximum depth of 15 was optimal for better accuracy.

ii) Data Bootstrapping: We also implemented data bootstrapping for making random forest trees. Each time we randomly chose one third of the total number of features to start with and then used those to create variety of different version of trees to increase randomness on tree features to improve accuracy. For data fraction to be selected for bootstrapping, we used 0.85 as fraction of sample size to be selected for training tree each time. This was also for dealing with highly skewed data.

iii) Number of Trees: We also used number of trees as another parameters to control the population of trees. Through the random choice of features in the trees, we tried to incorporate ensemble learning and randomness into the training process.

1. Short Description of the code:

The complete script implements a full end-to-end fraud detection from test data. It begins by loading transaction data, followed by cleaning and normalization of numerical attributes through 'processNumericData()'. Impurity measures such and information gain are employed to build a decision tree, with chi-square testing ensuring that only statistically meaningful splits are accepted. The trees are combined into a random forest ensemble, which improves generalization through random sampling (bootstrapping) and feature selection. Finally, predictions are evaluated with Kaggle's validation and accuracy was estimated based on that. Below are different sections of the code with snippets as screenshot for better reference inside the script.

2: The structure of the code:

2.1: Data Input

The script begins by loading the training and testing datasets using the Pandas library. The training dataset contains a column named 'isFraud', which acts as the target class variable for with values only 0 or 1, while the testing dataset is used later for model evaluation and predictions. The training and testing data is read into Pandas DataFrames named 'data' and 'testdata'. This is in line 8 and 9 of the script.

```
data = pd.read_csv("../Team Version/train.csv") # Load data from file and create a dataframe
testdata = pd.read_csv("../Team Version/test.csv") # Load data from file and create a dataframe
```

Fig. 2.1: Snippet for data input from the script

2.2 Data Preprocessing

The 'processNumericData()' function prepares numeric columns by handling missing values, converting data types, and addressing skewness or frequency imbalances in the feature distributions. It ensures that the data is numerically stable and ready for tree-based model training. The function first converts the string data into numeric data. When the function encounters "Not Found", it converts it to "Nan" which is being also used as a category for analysis. For this missing NaN data, the NaN was replaced with median between two adjacent values to smooth the data and convenient of statiscial analysis of data. The function uses hyperparameter "topFreqThresh" as input for quintile data binning put them into "nBins". This was adopted due the high skewness of the data.

The bins are returned as ranges for each bin and used as category boundaries for overall numerical data corresponding to the feature. In exceptional cases, where the unique values are very few, binning was skipped.

```
def processNumericData(data, numericFeatures, skew=1.0, topFreqThresh =0.5, nBins=5):
    processedData=data.copy()
    processedData=processedData.replace("NotFound", np.nan)
    for feature in numericFeatures:
        processedData[feature] = pd.to_numeric(processedData[feature], errors="coerce")

        #replace NAN with median values
        processedData[feature]=processedData[feature].fillna(processedData[feature].median())
        try:
            #Use Noramlized frequencies instead of normal freq
            topFreq=processedData[feature].value_counts(normalize=True, dropna=True).iloc[0] #in des
        except IndexError:
            continue
        if abs(processedData[feature].skew()) > skew or topFreq>topFreqThresh:
            #Using Quntile binning for highly skewed data, where top freq > 0.5
            try:
                processedData[feature]=pd.qcut(processedData[feature], q=nBins, duplicates='drop')
            except ValueError: #if the unique values are very few then skip binning
                processedData[feature]=processedData[feature]
        else:
            processedData[feature]=processedData[feature].astype(float)
    return processedData
```

Fig. 2.2: Snippet for data input from the script

2.3: Impurity Measures: Entropy, Gini, and Misclassification Error

To evaluate how well a split separates the target classes, the code defines three impurity measures—entropy, Gini-index, and misclassification error impurity assessment criteria. Each measure is adjusted to account for class imbalance by introducing an imbalance ratio (IR), defined as the ratio of minority to majority class frequencies. The weighting ensures that the algorithm does not overly favor the majority class when calculating impurity.

```
def entropy(data, target_attribute, IR, majorClass): # define entropy function using features and ta
    count=Counter(data[target_attribute]) # make count dict for determining number of instances of t
    minorClass=1 if majorClass==0 else 0
    total=count[majorClass]*IR+count[minorClass] # total instances
    entropy=0 # initialize entropy calculation
    for key, val in count.items():
        wt = IR if key==majorClass else 1
        prob=val*wt/(total) # calculate probability for each attributes
        # prob=val/total # calculate probability for each attributes
        entropy-=(prob)*math.log2(prob) # calculate entropy
    return entropy

def giniIndex(data, target, IR, majorClass): #define gini index function using data and attribute as
    # giniIndex= 1-Sub(P2)^2;
    count=Counter(data[target]) # Count values agains "isFraud" class
    minorClass = 1 if majorClass==0 else 0
    total = count[majorClass]*IR + count[minorClass] # total instances
    gini=1 # initialize weightedGiniIndex calculation
    for key, val in count.items(): # Loop for calulating gini
        wt = IR if key==majorClass else 1
        gini-=math.pow(val*wt/total, 2) # math for gini calculation
    print(gini)
    return gini

def misClassificationError(data, target, IR, majorClass):
    # error = 1-max(Pj (for all j))
    count =Counter(data[target])
    minorClass=1 if majorClass==0 else 0
    total=count[majorClass]*IR+count[minorClass] # total instances
    maxProb=0
    for key, val in count.items():
        wt = IR if key==majorClass else 1
        prob = val*wt/total
        maxProb=max(maxProb, prob)
    error =1-maxProb
    return error
```

Fig. 2.3: Snippet for Entropy, Gini Index and Misclassification Error

2.4. Information Gain – informationGain()

The 'informationGain()' function quantifies the effectiveness of a potential feature split in reducing impurity. It serves as the principal criterion for selecting the best split at each decision node. This function uses data and a certain feature as potential split for consideration and then check if the split is the best split. It also calculates impurity for child nodes and then based on the difference of IG values for parent and child node, returns overall IG for “bestSplit” function

```
def informationGain(data, feature, target, impurityMethod): #Define infromationgain function using data, feature and attribute as inputs

    #get unique values of the feature
    totalLen=len(data)
    if totalLen==0: return 0.0
    #Calculating Imbalance Ratio = countMinorityClass/countMajorityClass
    count=Counter(data[target])
    if 0 not in count or 1 not in count: return 0.0
    majorityClassCount=max(count[0], count[1])
    majorClass=0 if count[0]>count[1] else 1
    minorityClassCount=min(count[0], count[1])
    IR=minorityClassCount/majorityClassCount
    # calculate impurity for parent Node using entrogy, gini-index or misclassificaion error for IG calculation
    parentImpurity=entropy(data, target, IR, majorClass) if impurityMethod=="entropy" else (giniIndex(data, target, IR, majorClass) if in
    featureVal=data[feature] # get attibutes from feature
    print(featureVal)
    if pd.api.types.is_numeric_dtype(featureVal):
        sorted_values=np.sort(featureVal.unique())
        if len(sorted_values)<=1: return 0
        thresholds=(sorted_values[:-1] +sorted_values[1:])/2
        bestIG=-float("inf")

        for t in thresholds:
            left=data[featureVal<=t]
            right=data[featureVal>t]
            if len(left)==0 or len(right)==0:
                continue
            leftImpurity=entropy(left, target, IR, majorClass) if impurityMethod=="entropy" else (giniIndex(left, target, IR, majorClass)
            rightImpurity=entropy(right, target, IR, majorClass) if impurityMethod=="entropy" else (giniIndex(right, target, IR, majorCla

            weightedChildNodeImpurity=(len(left)/totalLen*leftImpurity + len(right)/totalLen *rightImpurity)
            IG=parentImpurity-weightedChildNodeImpurity
            bestIG=max(bestIG, IG)
        return bestIG
    uniqueVal=featureVal.unique()
    childImpurity=0.0
    for val in uniqueVal: ## Loop for IG calculation
        subData=data[data[feature]==val] # filter data for unique attribute
        subLen=len(subData)
        print(subData)
```

```
    childImpurity=0.0
    for val in uniqueVal: ## Loop for IG calculation
        subData=data[data[feature]==val] # filter data for unique attribute
        subLen=len(subData)
        print(subData)
        # calcualte impurity for unique feature and filtered data
        childNodeImpurity=entropy(subData, target, IR, majorClass) if impurityMethod=="entropy" else (giniIndex(subData, target, IR, majorClass
        childImpurity+=(subLen/totalLen)*childNodeImpurity # calculate information gain
    print(uniqueVal)
    IG=parentImpurity-childImpurity
    return IG
```

Fig. 2.4: Snippet for information gain from the script

2.5. Chi-Square Test – ChiSquare_test()

Before splitting on a feature, the algorithm uses a chi-square test to ensure that the split is statistically significant. This prevents the model from overfitting on random fluctuations in the data. The test returns 0 if the parent node is not a leaf node and returns 1 if the parent node is a leaft node. It uses alpha as input to test the null hypothesis. The alpha is also a hyperparameter for optimization.

```
def ChiSquare_test(df,Parentfeature,Childfeature,alpha): # Define function to do Chi-Square test using data, Parentfature and Childfeature, alp
    Parentunique = list(df[Parentfeature].unique()) # finding unique attributes in parent feature node
    childunique = list(df[Childfeature].unique()) # finding unique attributes in child feature nodes
    DOF = (len(Parentunique)-1)*(len(childunique)-1) # calculate degrees of freedom

    ChiMat = pd.crosstab(df[Parentfeature], df[Childfeature]).to_numpy() # calculate Chi-square count matrix
    rowsum = np.sum(ChiMat,axis=1,keepdims=True) # calcuate row sum matrix
    colsum = np.sum(ChiMat,axis=0,keepdims=True) # create column sum matrix
    T = np.sum(colsum) # calcuate total number of counts
    ChiMatExp=np.matmul(rowsum,colsum)/T # create expectation matrix
    Chi_sq=np.sum(((ChiMat-ChiMatExp)**2)/ChiMatExp)# calcuate Chi-square test statistics
    Critical_value = stats.chi2.ppf(1-alpha,DOF) # Find critical Chi-square value for given degree of freedom and alpha value
    test = 1 # initialize test

    if Chi_sq >= Critical_value: # conduct test by comparing with critical value
        test = 0 # Null hypothesis is rejected, DO NOT make parent node a leaf node
    else:
        test = 1 # Null hypothesis is accepted, make parent node as leaf node

    return test
```

Fig. 2.5: Snippet ChiSquare test from the script

2.6. Selecting the Best Split – bestSplit()

The 'bestSplit()' function introduces randomness by evaluating only a subset of features, which helps simulate the behavior of a random forest and prevents correlation among trees. The number of features taken each time is 1/3 of the total features. The function return best feature to be selected for further tree making and also best IG.

```
#slitting features
def bestSplit(data, features, targetValue, impurityCriteria):
    bestFeature=None
    bestIG=-1
    randomFeatures=np.random.choice(features, size=int(np.sqrt(len(features))), replace=False)
    for feature in randomFeatures:
        IG=informationGain(data, feature, targetValue, impurityCriteria) #IG method needs data, features, t
        print(IG)
        if(IG>bestIG):
            bestIG=IG
            bestFeature=feature
    return bestFeature, bestIG
```

Fig. 2.6: Snippet for choosing best split criteria from the script

2.7. Tree Node Definition – TreeNode Class

The 'TreeNode' class represents the fundamental structure of the decision tree. Each node can either be an internal decision point or a terminal leaf that provides a prediction.


```
#slitting features
def bestSplit(data, features, targetValue, impurityCriteria):
    bestFeature=None
    bestIG=-1
    randomFeatures=np.random.choice(features, size=int(np.sqrt(len(features))), replace=False)
    for feature in randomFeatures:
        IG=informationGain(data, feature, targetValue, impurityCriteria) #IG method needs data, features, t
        print(IG)
        if(IG>bestIG):
            bestIG=IG
            bestFeature=feature
    return bestFeature, bestIG
```


Fig. 2.7: Snippet for choosing best split criteria from the script

2.8. Recursive Tree Construction – constructTree()

The recursive function 'constructTree()' builds the decision tree from the root node down to the leaves. It uses the impurity measures, information gain, and chi-square significance to determine where and when to split and return a node for each iteration

```
#Tree Construction
def constructTree(data, features, target, impurityCriteria, alpha, depth=0, maxDepth=None):
    data = data.copy()
    data=data.dropna(how='all')
    count =Counter(data[target])
    majority=data[target].mode()[0]
    #if pure (all rows have same class)
    if len(data[target].unique())==1:
        return TreeNode(leafPrediction=data[target].iloc[0], class_counts=count)
    #if no feature left/small data
    if not features or (maxDepth and depth>=maxDepth):
        return  TreeNode(leafPrediction=majority, class_counts=count)

    #Find the best split using the information Gain
    bestFeature, IG=bestSplit(data, features, target, impurityCriteria)
    #check chi-square for split and to decide further splitting
    chiSquare=ChiSquare_test(data, bestFeature, target, alpha)
    if(chiSquare):
        return TreeNode(leafPrediction=majority, class_counts=count)
    #otherwise expand the tree Node
    node =TreeNode(featureAtNode=bestFeature, leafPrediction=majority, class_counts=count)
    for f in data[bestFeature].unique():
        subset=data[data[bestFeature]==f]
        if subset.empty:
            majorityValOfParentNode=data[bestFeature].mode()[0]
            node.children[f]=TreeNode(leafPrediction=majorityValOfParentNode, class_counts=count)
        else:
            remainingFeatures= [f for f in features if f!=bestFeature]
            node.children[f]=constructTree(subset, remainingFeatures, target, impurityCriteria, alpha, dept
    return node
```

Fig. 2.8: Snippet for constructing tree for the Random Forest algorithm from the script

2.9. The Random Forest Model – randomForest Class

The randomForest class implemented the ensemble learning component of the script, combining multiple decision trees to produce a more stable and accurate predictive model. The forest builds on decision trees which were randomly feature-selected and was constructed using impurity-based splitting criteria, and introduces randomness both in data sampling and feature selection to enhance diversity among individual trees.

When initialized, the class stores all the necessary parameters, including the dataset, feature list, number of trees to build, sampling fraction, target variable name, impurity calculation method (e.g., Gini-index or entropy), chi-square significance level, and the maximum depth of each tree. The ensemble of trees is maintained in a list attribute named self.trees, which is populated during the training process.

The training phase is implemented through the randomTree() method. This method performs bootstrap sampling, where each tree is trained on a random subset of the original data selected with replacement. This ensures that every tree is exposed to slightly different samples, reducing the likelihood of overfitting to specific data points. Additionally, at each node of the decision tree, only a random subset of features—approximately the square root of the total number of available features—is considered for splitting. This process is handled implicitly through the bestSplit() function and ensures that the trees do not become overly correlated by repeatedly using the same dominant features. Each resulting tree is constructed via the constructTree() function and stored within the forest's internal list. Prediction is achieved through the predictTree() method. For each observation (or test instance), the method traverses every decision tree from root to leaf, following the appropriate feature splits based on the instance's attribute values. The traversal continues until a terminal node is reached, at which point the leaf's class prediction is recorded. Once all trees have made their individual predictions, the final class label for that instance is determined through a majority voting scheme, where the most common predicted label across all trees is chosen as the ensemble's output. This approach substantially improves predictive performance compared to a single decision tree. The use of bootstrapping and random feature selection reduces variance, mitigates overfitting, and ensures that no single feature or subset of the data dominates the learning process. As a result, the randomForest class provides a balanced trade-off between model interpretability and generalization power, making it particularly effective for high-dimensional or imbalanced classification problems, such as fraud detection.

```
class randomForest:
    def __init__(self, data, features=None, numberOfTrees=None, sampleSize=None, target="isFraud", impurity
        self.data=data
        self.features=features
        self.numberOfTrees=numberOfTrees
        self.sampleSize=sampleSize
        self.target=target
        self.impurityCriteria=impurityCriteria
        self.alpha=alpha
        self.depth=0
        self.maxDepth=maxDepth
        self.trees=[]

    def randomTree(self):
        for i in range(self.numberOfTrees):
            sample = self.data.sample(frac=self.sampleSize, replace=True, random_state=None)
            randomFeatures=list(np.random.choice(self.features, size=max(1, int(np.sqrt(len(self.features))
            tree=constructTree(sample, randomFeatures, target=self.target, impurityCriteria=self.impurityCr
            self.trees.append(tree)

    def predictTree(self, instance):
        randomTreeResult=[]
        randomTrees=self.trees
        for rTree in randomTrees:
            node =rTree
            while not node.isLeaf():
                feat=node.featureAtNode
                val=instance[feat]
                if pd.isna(val):
                    val = "Missing"
                if val in node.children:
                    node =node.children[val]
                else:
                    break
            randomTreeResult.append(node.leafPrediction)
        majorityVote=Counter(randomTreeResult).most_common(1)  #[(class, occurance)]
        print(majorityVote)
        return majorityVote[0][0]
```

```
    def predictAll(self, testData):
        preds =[]
        for _, instance in testData.iterrows():
            preds.append(self.predictTree(instance))
        # test_preds = pd.Series(preds, index=testdata.index, name="predicted")
        output = pd.DataFrame({
            "TransactionID": testdata["TransactionID"],
            "predicted": preds
        })
        output.to_csv("sampledata.csv", index=False)
        print(preds)
```

Fig. 2.9: Snippet for making random forest from trees produced from “ConstructTree” function from the script

2.10. Hypoerparameters and their functions:

The script uses number of hyperparameters which serves different purposes:

numberOfTrees: This is to control number of trees to be included in forest. It basically controls the population of trees to trade off between accuracy vs computational time.

sampleSize: This parameter was used to chose the fraction of data for bootstrapping. The fraction was chosen to be part of whole training dataset for “bagging” with replacement. This affects the randomness of the features each time a tree is constructed and controls the training also. Together with random choice of features, randomness or ensemble training was achieved.

ImpurityCriteria = IG methods chosen to compare accuracy and structure of the algorithm

Alpha = This parameter was used for Chi-square test. This was used to control the decision on whether a parent node can be a leaf node based on alpha as a critical value of termination of a node.

MaxDepth: This was used to control depth of the trees. Too much depth of trees are prone to cause overfitting of data and algorithm may be computationally expensive. This parameter was used to get optimal depth to have reasonable accuracy and data testing time.

Appendix: Data Analysis

Data Analysis:

The main purpose of data analysis is to select a set of features that will be used to make decision trees that constitute random forest. In this effort, the first feature was “TransactionID” which was discarded since every transaction has unique ID and it is serving as mere index for data recording. Among all the features, only three feature was found to be completely categorial: “ProductCD”, “card4”, “card6”. Those were taken into account immediately for budling decision tree. The rest of the features are numerical by looking at a first glance. We broadly divided the rest of the numerical ones into three broad categories to be analyzed: “TransactionAmt” ,“TransactionDT” into transaction category; “addr1” and “addr2” into address category; “card1”, “card2”, “card3” and “card5” into card category and “c1”-“c14” into C-data category. The C-data are highly numerical and assumed to be coded. For the first attempt, we tried to first correlate the data using using “.corr” library of Pandas. The script “Correlation_check.py” includes simple script for this. We created a “heatmap” among all the features as shown in Fig. 1.

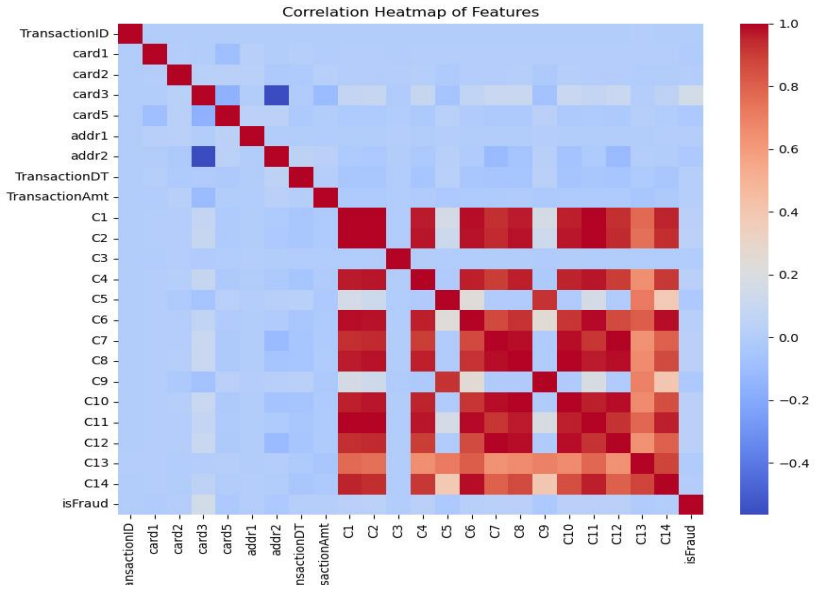


Fig. 1: Correlation Heatmap of features for data provided in “train.csv”

As shown in Fig. 1, the diagonal squares all represent autocorrelation of the features and can be discarded for analysis. As the data shows, from “C1” to “C14” the features are mostly correlated except for the column “C3”. “addr2” is correlated to “card3” which are both taken into consideration for data analysis and splitting. The “TransactionAmt” was found to be similar to “TransactionID” in a sense that it has 7,898 unique counts that can be very difficult to categories into broad categories with transaction amounts vary in ranges from two digits (e.g. 100$) to three to four digits (e.g. $300 or $4000). The “TransactionDT” was similar in category like “TransactionAmt” with even 461340 unique counts for time deltas. For rest of the numerical features i.e. “card1”, “card2”, “card3”,“card5”,”addr1” ,“addr2”, “C1,C2,…..C14”, we tried to visualize data concentration i.e. highest value counts for particular class as a function of numerical features. The purpose of this is to find a subset of feature variables corresponding to a specific feature (e.g. “C2”) from the whole set of data into different sections, where we could discretize numerical features into categories (e.g. “C2”>10 or “C2”<10… etc.). In other words, we tried to visualize P(“C2”=0,1,2,3…|”isFraud”=0) and P(“C2”=0,1,2,3…|”isFraud”=1) into a bar plot to find a range where we could split the data for that feature. Below are list of bar plots we produced to analyze the data.

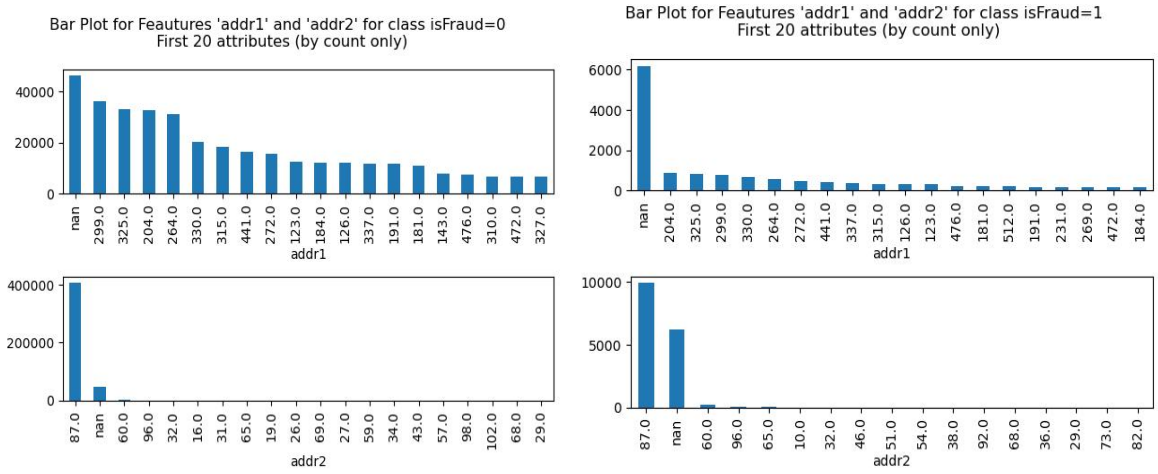


Fig. 2: Bar plot showing “addr1” and “addr2” features. We used value_count() function from Pandas library to count unique frequency counts and plot the first 20 of the attributes to show frequency of most occurrences and a range.

For address features, first we used NaN to separate and isolate counts for “Not Found” address entries. We found that in addr2 “87” and “Not Found” has highest frequencies of occurrence. So, we used them as attributes for decision tree.

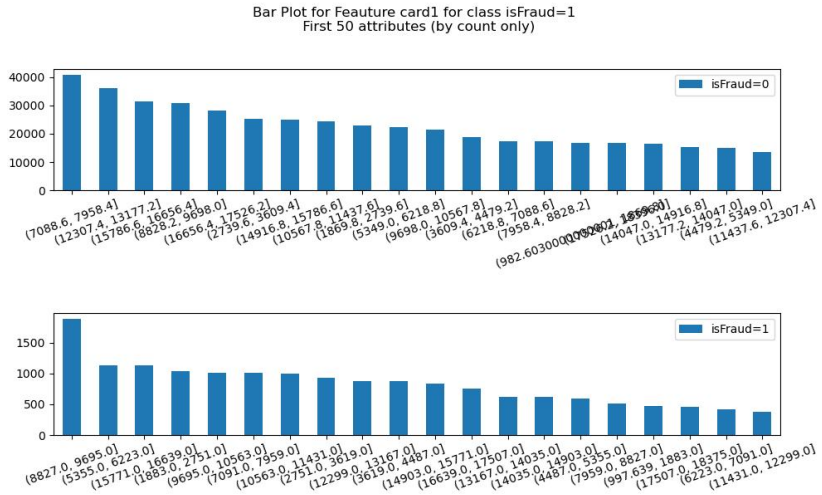


Fig. 3: Binned bar plot for “card1” feature. The number of high frequency feature in this is too high.

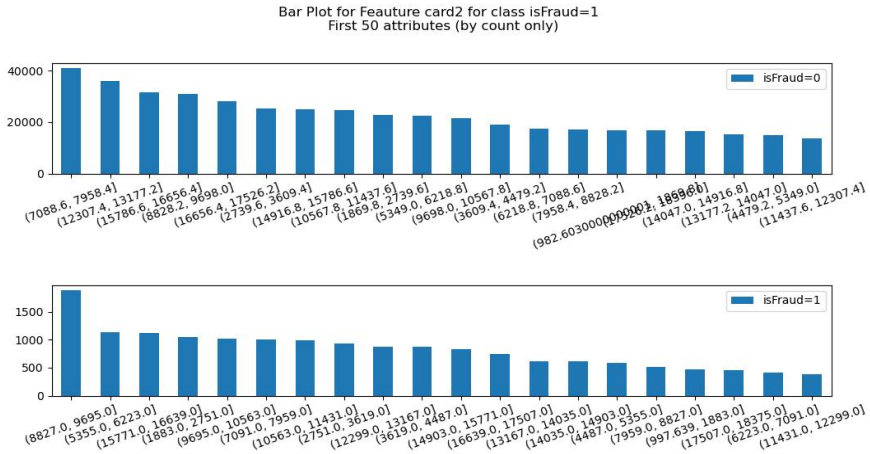


Fig. 4: Binned bar plot for “card1” feature. The plot was used to identify numbers where to split the feature at “card2”

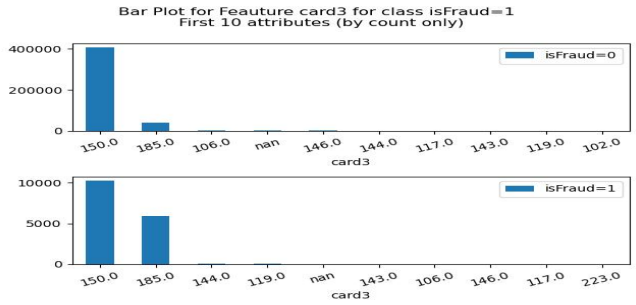


Fig. 5: Bar plot for “card3” feature. We found that 150 and 185 are highest occurring attributes for this feature.

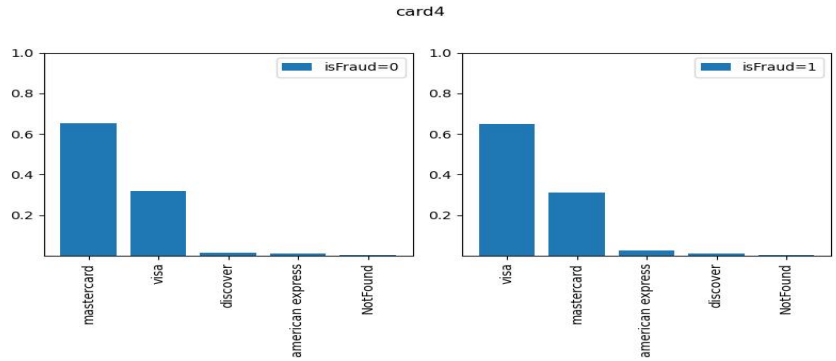


Fig. 6: Normalized Bar plot for “card4” feature. We found that 150 and 185 are highest occurring attributes for this feature.

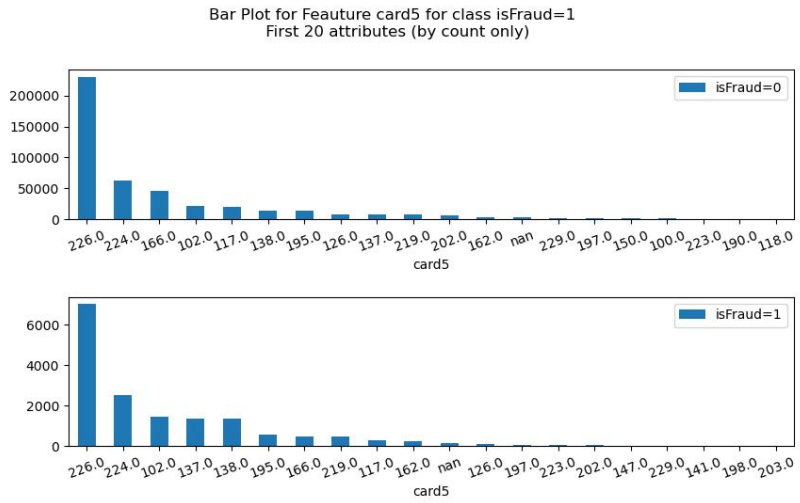


Fig. 5: Bar plot for “card5” feature. We found that some features such as 102, 117, 137, 138, 166, 195, 226, 224 are highest occurring attributes for this feature.

For different cards, “card1” to “card5” . The list of attributes found to be useful is:

Card Number	Attributes
card1	Discarded, too many attrbiutes to be categorized
card2	10,000>Card2>=9000 and 16,000>Card2>=15000
card3	150, 185
card4	Mastercard, Visa, American Express, Not Found
card5	102, 117, 137, 138, 166, 195, 226, 224