CS529 Project 1: Random Forests

[Caleb Annan, MingChe Pei, Dawud Shakir, Sina Mokhtar]

March 09, 2024 [Table 5 Corrected March 15, 2024]

1 Introduction

We use a collection of decision trees to make a forest majority prediction of the competition dataset. Ultimately our best submission received a 72% accuracy on Kaggle.

Random forests are a collection of small decision trees. They are built using small, randomly selected samples. Forests are discriminatory (not generative) learners and are iteratively built. They become better at predicting after each iteration.

In a forest, each tree is built (or "fitted") with random samples from a dataset. The, one random sample is selected from our dataset for testing and each tree makes one prediction: 0 or 1 (False or True). The forest majority's prediction is compared to the actual value. Samples that cause the majority to give an incorrect value are *more likely* to be chosen in subsequent iterations.

After a number of iterations, the forest majority becomes better at predicting than a single decision tree.

Each decision tree is built using features (attributes) and a target from the training dataset. Using Python3 with Pandas, we preprocess the training and testing dataset, we remove columns and subcolumns without meaningful information (iterative columns, χ^2 testing), we replace or drop missing values, we threshold continuous variable data. After preprocessing, we model decision trees individually and collectively (forests), we use hyperparameters to optimize model performance, and evaluate our model with accuracy, precision, recall, and F1-score.

In this project, we learnt about datasets, features, model interpretation, and decision making. We learnt prepping techniques to handle missing values, convert continuous data to nominal values, feature selection strategies, and model tuning to improve predictive performance.

2 Data Description

Our decision trees are trained using two datasets: a feature dataset and a target dataset.

TransactionID	ProductCD	card1	card2	card3	card4	card5	card6	addr1	addr2	TransactionDT	TransactionAmt	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	isFraud
1	C	5812	408.0	185.0	mastercard	224.0	debit	NotFound	NotFound	13184610	22.14	1.0	1.0	0.0	1.0	0.0	1.0	1.0	1.0	0.0	0.0	1.0	1.0	1.0	1.0	1.0
0																										
2	W	12577	268.0	150.0	visa	166.0	debit	123.0	87.0	12274010	107.94	141.0	124.0	0.0	0.0	123.0	94.0	0.0	0.0	85.0	0.0	104.0	0.0	457.0	120.0	0
3	H	12581	453.0	150.0	visa	226.0	debit	272.0	87.0	1343827	75.00	1.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	1.0	1.0	1.0	0	
4	W	8139	327.0	150.0	discover	194.0	credit	204.0	87.0	15034938	92.00	6.0	1.0	0.0	0.0	0.0	2.0	0.0	0.0	2.0	0.0	3.0	0.0	18.0	6.0	0
5	W	16132	111.0	150.0	visa	226.0	debit	220.0	87.0	14680707	39.00	1.0	1.0	0.0	0.0	0.0	2.0	0.0	0.0	1.0	0.0	1.0	0.0	28.0	1.0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
472428	W	9622	562.0	150.0	mastercard	102.0	credit	143.0	87.0	3430616	226.00	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	2.0	0.0	1.0	0.0	3.0	1.0	0
472429	W	7826	481.0	150.0	mastercard	224.0	debit	325.0	87.0	9110536	24.50	5.0	3.0	0.0	0.0	0.0	3.0	0.0	0.0	1.0	0.0	1.0	0.0	13.0	5.0	0
472430	W	8868	205.0	150.0	visa	166.0	debit	327.0	87.0	8646323	59.00	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	1.0	0
472431	W	8320	476.0	150.0	visa	166.0	debit	330.0	87.0	7333259	59.00	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	1.0	0
472432	H	6019	583.0	150.0	visa	226.0	credit	420.0	87.0	1280322	50.00	1.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	1.0	0.0	1.0	1.0	0

Table 1: The training dataset (train.csv) has 472,432 samples. Each sample is a row. The last column is the target and the remaining columns are features.

Index	Column Name	Number of Unique Values
0	ProductCD	5
1	card1	12815
2	card2	501
3	card3	112
4	card4	5
5	card5	115
6	card6	5
7	addr1	304
8	addr2	72
9	TransactionDT	461340
10	TransactionAmt	7898
11	C1	1350
12	C2	1064
13	C3	21
14	C4	1078
15	C5	319
16	C6	1158
17	C7	914
18	C8	1002
19	C9	204
20	C10	987
21	C11	1210
22	C12	927
23	C13	1393
24	C14	1018
25	isFraud	2

Table 2: Unique Values in Each Column

The last column "isFraud" is the target dataset. It gives a binary classification "0" or "1" ("not fraud" or "fraud") for each sample (row) in the dataset. The first column "TransactionID" is for indexing. The 25 remaining columns are the feature dataset. Each column in the feature dataset is either continuous or categorical. Continuous values are always numbers and categorical values can be either numbers or letters. "NotFound" means a value is missing in a column.

Before pre-processing, there are 455,902 zeros and 16,530 ones (97%/3%) in the entire 472,432 training samples. This is an imbalance ratio of,

$$\text{Imbalance Ratio (IR)} = \frac{\text{number of ones}}{\text{number of zeros}} = \frac{3}{97} \approx 0.031.$$

The target dataset is imbalanced: there are many more zeros than ones. Since the data is more skewed towards zeros than ones, predictions decision trees built with the dataset will be biased. In the real world, that could possibly predict a fraudulent transaction when there is not one (false negative).

Here is the entire dataset grouped by the isFraud column (front). The column in the end shows the number of duplicate samples with either "0" or "1" classification.

3 Decision Trees and Forests (Oh My!)

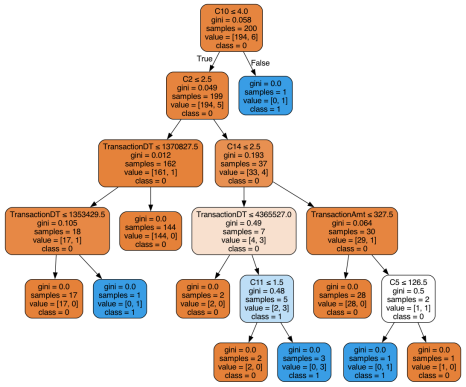

Figure 1: A 2-branch decision tree generated using graphviz (see appendix). Blue nodes are leafs nodes. Brown nodes are decision nodes. This tree was generated using Python and GraphViz.

This section gives an overview of the model building process for decision trees and random forests.

3.1 Random Forests

In a random forest, subsets of the training dataset are sampled randomly to build a collection of small, decision trees. Then one sample is given to each decision tree and each decision tree predicts either a "0" or a "1". The majority vote is compared with the "correct" prediction.

Samples that produced more incorrect samples are weighed higher in subsequent iterations.

3.2 Decision Trees

The set of all nodes underneath node x is called the sub-tree rooted at x. The size of a tree is its number of nodes; a leaf by itself has size 1.

Decision trees are models used for classifying samples. Here, our decision tree is used to classify samples as either "0" or "1" implying a non-fraudulent transaction or a fraudulent one, respectively. Our decision tree uses different impurity calculation techniques, i.e., entropy, gini index, misclassification error, in building our tree.

Decision trees are tree-like structures. The topmost node is called the *root* node. The root node has branches (edges) that descend down to *child nodes* or to a "0"-leaf or "1"-leaf. Each child node is also a sub-tree with branches descending down to other nodes and leaves. A tree is built iteratively or recursively by partitioning the feature (column) dataset. If a node has no branch (*all* zeros or *all* ones), it becomes a leaf (*terminal node*) and assigned a "0" or "1".

Each node represents a feature, each branch represents a decision for that feature, and each leaf node represents an outcome or prediction. Each partition or split puts the feature with the best split of the data at each node.

Because of conditional independence between columns, all columns are split by a given *row*. Values in a row with "0" ... Values in a row with "1". What this means is, in a given column, there are distinct characters or values. Each unique value corresponds to a branch of the feature or node. Furthermore, as these branches lead to a '0' or '1' at the leaf node, a certain percentage of the unique values will correspond to '1', while the remaining percentage will correspond to '0'. This distribution of values and the associated impurity are quantified by measures such as entropy, the Gini index, or the misclassification error.

3.3 Implementation

We use the C4.5 Decision Tree because it can handle both continuous numerical and categorical data.

3.4 Thresholding

Columns with non-categorical variable data are thresholded. The brute force method "splits" a column by every number in its column. Each split creates two branches with the number as the threshold. Numbers less than or equal to that threshold ("<= 80.0") are sorted to the "left", numbers more than that threshold ("> 80.0") are sorted to the "right". The split with the best information gain decides the information gain for that column (feature).

The percentile method is the same as brute force except instead the 1st, 2nd, ..., 100th percentiles of the column are used as thresholds to divide the column's data.

After finding the threshold to split the numbers in the column, each value is converted to a string. Non-categorical columns with numbers become 2-

branched categorical nodes. For instance, if 80.0 is chosen as the threshold, every number in that column becomes either "<= 80.0" or "> 80.0".

3.5 Sample Selection

We trained our model with the entire training dataset (100%), half the training dataset (50%) and a quarter of the training dataset (25%).

We used oversampling method to solve the class imbalance problem. This can increase the number of minority class samples and make the number of samples in different classes more balanced.

We also used undersampling method, but the result was a little bit worse than using oversampling.

3.6 Hyperparameters

We used these hyperparameters while training and re-training our models:

- maximum depth of the tree (`max_depth`)
- maximum depth reached while building the tree (`max_depth_reached`)
- minimum number of samples required to split a node (`min_samples_split`)
- minimum number of samples required to be at a leaf node (`min_samples_leaf`)

3.7 Basic Decision Tree Algorithm

Our basic algorithm is based on Mitchell's description of decision tree construction in his book "Machine Learning".

Features with categorical [1] values have many-branches.

Features that are only numbers are 2-branches.

3.8 Finding the Best Split

We find the best column to split at each level of our decision tree by calculating the potential gain of splitting each column. We use the column with the best information gain. The formula for information gain is:

$$\text{Information Gain}(S, A) = \text{Impurity}(S) - \sum_{v \in \text{values}(A)} \frac{|S_v|}{|S|} \text{Impurity}(S_v)$$

Where the samples are S, A is the column being split, values(A) are A's values, $|S|$ is the number of samples, and $|S_v|$ is the number of values in column A with value v.

After the information gain is calculated for each column, and the feature with the highest information gain is selected as the best feature to split between values that are "0" and values that are "1".

[1] Or "nominal" meaning "name only".

3.8.1 Impurity

$$\text{entropy} = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

$$\text{gini index} = 1 - \sum_{i=1}^{n} p_i^2$$

$$\text{misclassification error} = 1 - \max(p_1, p_2, ..., p_n)$$

3.9 χ^2 Significance

We use chi-squared to decide whether splitting a node would be meaningful (significant) or whether there is not enough information. If there is not enough information, then it becomes a leaf with the most common value in the target datatset. The chi-square test statistic (χ^2) is calculated as follows:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

where:

- O_i = Observed frequency count for category i
- E_i = Expected frequency for category i
- The sum of χ^2 for all categories.

Once we find the χ^2 value, we compare it to the critical value (p-value) from the chi-square distribution table. To find the p-value, we want the degrees of freedom and a confidence level:

$$\text{degrees of freedom} = \text{number of nodes} - 1 \cdot (\text{number of classes} - 1)$$

We use confidence values of 0.85, 0.90, 0.95 and 0.97.

We use stats.chi2 in Python to look up the p-value. If the p-value is more than χ^2, the node's information (the frequency counts for values) is the same for the node and its parent node. This means continuing would not be significant. We do not continue building that node. We then turn the node into a leaf and make the most common value (mode) that leaf's value.

3.10 Evaluation

We evaluated our model's performance using accuracy, precision, recall, and F1-score.

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

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

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

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

4 Experimentation

Table 3: Entropy Metrics

Alpha	Confidence	Accuracy	False 1 Score
0.01	0.99	0.83	0.18
0.05	0.95	0.77	0.15
0.10	0.90	0.96	0.00
0.25	0.75	0.04	0.07
0.50	0.50	0.97	0.00
0.90	0.10	0.03	0.07

Table 4: Gini Metrics

Alpha	Confidence	Accuracy	False 1 Score
0.01	0.99	0.75	0.14
0.05	0.95	0.77	0.15
0.10	0.90	0.00	0.00
0.25	0.75	0.04	0.07
0.50	0.50	0.97	0.00
0.90	0.10	0.03	0.07

5 Prepping Data

5.1 Iterative columns

Iterative columns (0, 1, 2, 3, ...) do not provide meaningful information. They were removed.

5.2 Missing values

Missing values are represented with "NotFound".[2]

Index	Column	"NotFound"	Percentage (%)
0	TransactionID	0	0.00
1	ProductCD	0	0.00
2	card1	0	0.00
3	card2	7203	6.04
4	card3	1258	1.06
5	card4	1269	1.06
6	card5	3404	2.85
7	card6	1261	1.06
8	addr1	52428	43.98
9	addr2	52428	43.98
10	TransactionDT	0	0.00
11	TransactionAmt	0	0.00
12	C1	0	0.00
13	C2	0	0.00
14	C3	0	0.00
15	C4	0	0.00
16	C5	0	0.00
17	C6	0	0.00
18	C7	0	0.00
19	C8	0	0.00
20	C9	0	0.00
21	C10	0	0.00
22	C11	0	0.00
23	C12	0	0.00
24	C13	0	0.00
25	C14	0	0.00
26	isFraud	0	0.00
Total		**119251**	**100.00**

We tried four approaches for dealing with missing values in the original dataset:

- We drop rows (samples) with a missing value in any column (119,251 rows).
- We drop columns that are more than 90.0% missing. No columns were missing more than 90%, although "addr1" and "addr2" were missing 43% and 43% of their values.

[2] If "NaN" was used, it was "not found!" lol.

- We impute missing values for each column (both strings and numbers) by using the mode (most common value).
- We used the mean to fill the missing value when missing value is less than 10%. We used the median to fill the missing values. Both results are almost the same. The mean yielded a higher public score on Kaggle.
- We treat the representation for missing values "NotFound" as a legitimate categorical value.

Index	**Column**	**With "NotFound"**	**Without "NotFound"**
0	TransactionID	472432	411799
1	ProductCD	5	5
2	card1	12815	11092
3	card2	501	495
4	card3	112	87
5	card4	5	4
6	card5	115	92
7	card6	5	4
8	addr1	304	299
9	addr2	72	70
10	TransactionDT	461340	403145
11	TransactionAmt	7898	5748
12	C1	1350	726
13	C2	1064	703
14	C3	21	21
15	C4	1078	399
16	C5	319	316
17	C6	1158	593
18	C7	914	84
19	C8	1002	229
20	C9	204	204
21	C10	987	322
22	C11	1210	611
23	C12	927	97
24	C13	1393	966
25	C14	1018	553
26	isFraud	2	2
Total		**119251**	**109997**

It took 290.0 seconds using 100% of the training samples (a train-test 100%-0% split) to build one tree with no max depth. It took 234 seconds to build and 19.94 to predict (train-test 80%-20% split) of training data.

Column	**Gain (Entropy)**	**Gain (Misclassification Error)**	**Gain (Gini Index)**
TransactionID	4.962111×10^{-7}	5.551115×10^{-17}	4.645328×10^{-8}
ProductCD	1.485583×10^{-2}	2.081668×10^{-17}	1.921272×10^{-3}
card1	4.889996×10^{-2}	1.242507×10^{-3}	8.447085×10^{-3}
card2	2.418911×10^{-2}	1.457168×10^{-16}	3.516869×10^{-3}
card3	1.422834×10^{-2}	2.328377×10^{-5}	2.075254×10^{-3}
card4	3.500386×10^{-4}	4.163336×10^{-17}	4.294809×10^{-5}
card5	8.927706×10^{-3}	2.116707×10^{-6}	1.061225×10^{-3}
card6	6.485255×10^{-3}	6.938894×10^{-18}	6.929912×10^{-4}
addr1	1.534362×10^{-2}	6.350120×10^{-6}	2.061978×10^{-3}
addr2	1.376107×10^{-2}	4.656755×10^{-5}	1.960744×10^{-3}
TransactionDT	4.195930×10^{-5}	0.000000×10^{-0}	3.927058×10^{-6}
TransactionAmt	3.683303×10^{-5}	2.775558×10^{-17}	3.447430×10^{-6}
C1	2.485583×10^{-3}	6.938894×10^{-18}	2.325848×10^{-4}
C2	3.430610×10^{-3}	4.163336×10^{-17}	3.197757×10^{-4}
C3	1.629332×10^{-4}	$-6.938894 \times 10^{-18}$	8.878662×10^{-6}
C4	1.236895×10^{-2}	4.857226×10^{-17}	1.388197×10^{-3}
C5	1.052007×10^{-2}	9.020562×10^{-17}	8.066106×10^{-4}
C6	1.830050×10^{-3}	2.081668×10^{-17}	1.793112×10^{-4}
C7	1.284657×10^{-2}	6.245005×10^{-17}	1.757447×10^{-3}
C8	1.119364×10^{-2}	2.081668×10^{-17}	1.239652×10^{-3}
C9	1.504398×10^{-3}	9.020562×10^{-17}	1.300779×10^{-4}
C10	1.105375×10^{-2}	2.081668×10^{-17}	1.237102×10^{-3}
C11	3.468303×10^{-3}	8.326673×10^{-17}	3.439170×10^{-4}
C12	1.178123×10^{-2}	8.326673×10^{-17}	1.434546×10^{-3}
C13	1.852612×10^{-3}	6.938894×10^{-17}	1.684108×10^{-4}
C14	5.405058×10^{-4}	7.632783×10^{-17}	4.944579×10^{-5}

Table 5: Gains for each feature using entropy, misclassification error, and gini index. The average time to build each column was 43.1 seconds. Gain values for binary classification are within the range of $0 \leq$ **Gain** ≤ 1. The lone negative value observed under **Misclassification Error** for "C3" is likely rounding error. Implementation available at https://github.com/dawud-shakir/DecisionTree/blob/main/gains.py.

5.3 Discussion

The one-tree model collects the chance of 1 for each value in the column. This method scored higher (70% accuracy) than our majority vote model (52% accuracy). This classifier was based on basic machine learning concepts such as entropy for impurity calculations and also information gain for determining best feature to use at a node in a decision tree, which is the basis of random forests. In our bid to build this classifier, we encountered classic data structures problems, ie., missing values and having continuous data which needs to be converted to nominal data for optimization. For missing data we used simple statistics-mode to fill them out if they were less than 10 percent of the entire column data. In converting continuous data to numerical data, we employed the C4.5 decision tree method to find a threshold which then basically splits all entries as greater than or less than. Instead of using the typical brute force method of finding the threshold between values, we used the percentiles approach which was more effective.

6 Conclusion

In this project, we have used the random forests method to build a classifier which is aimed at recognizing fraudulent transactions. Our best submission on Kaggle was a one-tree using just one column.

7 Deliverables

- Project Report
- Github Repositories:
  - `https://github.com/dawud-shakir/RandomTree`
  - https://github.com/caleb-annan/RandomTree
  - https://github.com/ming-che-pei/RandomTree
  - https://github.com/sina-mokhtar/RandomTree

Appendix: demo_basics.py

```
import numpy as np
import pandas as pd
import os
import time

### dataset ############################
```

```
categorical = (
    "ProductCD",
    "card1",
    "card2",
    "card3",
    "card4",
    "card5",
    "card6",
    "addr1",
    "addr2"
)

continuous = (
    "TransactionDT",
    "TransactionAmt",
    "C1",
    "C2",
    "C3",
    "C4",
    "C5",
    "C6",
    "C7",
    "C8",
    "C9",
    "C10",
    "C11",
    "C12",
    "C13",
    "C14"
)

target = (
    "isFraud"
)

### accuracy ##################################################################

def imbalance_ratio(y:pd.Series):
    counts = y.value_counts()
    minor = counts[1]
    major = counts[0]
    return float(minor / major)


def score(y, predictions):
    if len(y) != len(predictions):
```

```
        exit(f'y and predicted are different lengths: [{len(y)}, {len(prediction

    y = list(y)
    predictions = list(predictions)


    n_correct_0 = n_wrong_0 = n_wrong_1 = n_correct_1 = 0

    # calculate confusion matrix values
    for i in range(len(y)):
        if y[i] == predictions[i]:
            if y[i] == 0:
                n_correct_0 += 1
            else:
                n_correct_1 += 1
        else:
            if y[i] == 0:
                n_wrong_0 += 1
            else:
                n_wrong_1 += 1

    return n_correct_0, n_wrong_0, n_correct_1, n_wrong_1


def confusion_matrix(y, predictions):
    correct_0, wrong_0, correct_1, wrong_1 = score(y, predictions)
    return [[correct_0, wrong_0], [wrong_1, correct_1]]


def accuracy(y, predictions):

    correct_0, wrong_0, correct_1, wrong_1 = score(y, predictions)
    N = correct_0 + wrong_0 + wrong_1 + correct_1

    accuracy = 1 - ((wrong_0 + wrong_1) / N)
    weighted_accuracy = 1 - (0 * correct_0 + (1 * wrong_0 + imbalance_ratio(y) *

    return accuracy, weighted_accuracy

def standard_error(y, predictions):
    acc, weighted_accuracy = accuracy(y, predictions)
    N = len(y)

    standard_error = np.sqrt((1 / N) * acc * (1 - acc))
    weighted_error = np.sqrt((1 / N) * weighted_accuracy * (1 - weighted_accurac
```

```
    return standard_error, weighted_error


### missing values ###########################################################

def remove_iterative(df:pd.DataFrame, columns_to_remove):
    print(f"removing iterative columns {columns_to_remove}")
    df = df.drop(columns_to_remove, axis=1)
    return df

def impute_missing_data(df:pd.DataFrame):
    print("imputing missing data")

    # use not-a-number representation that is compatible with pandas
    df = df.replace(['NotFound', 'NaN'], float('nan'))


    nan_count_per_column = df.isna().sum()
    nan_count_total = df.isna().sum().sum()



    print(f"imputing {nan_count_total} values")

    # impute missing values with most common value in column
    for i in df:
        df[i] = df[i].fillna(df[i].mode()[0])



    return df

def drop_missing_data(df):
    print("dropping missing data")
    # use not-a-number representation that is compatible with pandas
    df = df.replace(['NotFound', 'NaN'], float('nan'))

    # drop rows
    df = df.dropna(axis=0)
    return df


### impurity #################################################

def purity(df_test, column_name):
```

```
    target_column = df_test[column_name]
    unique_classes = np.unique(target_column)
    if int(len(unique_classes))==1:
        return True
    else:
        return False


def gini_index(y):
    _, counts = np.unique(y, return_counts=True)
    probabilities = counts/len(y)
    gini_i = 1 - np.sum(probabilities**2)
    return gini_i


def calculate_entropy(y):
    _, counts = np.unique(y, return_counts=True)
    probabilities = counts / counts.sum()
    entropy = -np.sum(probabilities * np.log2(np.maximum(probabilities, 1e-10)))

    return entropy

### splitters ##############################################################

def best_split_threshold(df, column, impurity):


    if False:
        # brute force
        orderedvals = df[column].sort_values().values

    #percentiles
    prctls = df[column].describe(percentiles=np.array(range(1,100))/100)

     # exclude count, std, and mean
    orderedvals = [prctls[i] for i in prctls.index if ~np.isin(i, ['count','mean


    min_split = float('inf')
    best_valsmids = None

    for i in range(len(orderedvals)-1):
        if False:
            # with mean-stepping!
            valsmids = (orderedvals[i] + orderedvals[i+1])/2
```

```
        valsmids = orderedvals[i]

        databelow = df.loc[df[column] <= valsmids, column]
        dataabove = df.loc[df[column] > valsmids, column]

        B = impurity(databelow)
        A = impurity(dataabove)

        split = (len(databelow)/(len(databelow) +len(dataabove)))*B + (len(dataa

        if split < min_split:
            min_split = split
            best_valsmids = valsmids

    return best_valsmids


def numerical_split(df, column, best_valsmids):
    splits = {}

    databelow = df.loc[df[column] <= best_valsmids, column]
    dataabove = df.loc[df[column] > best_valsmids, column]

    splits['databelow'] = databelow
    splits['dataabove'] = dataabove

    return splits



#splits = numerical_split(df_test, 'card1', 12000)
#print(splits['databelow'])


def categorical_split(df, column):
    featurevals = df[column]
    branches = {}

    for i in featurevals.unique():
        branches[i] = df.loc[df[column] == i, column]

    return branches



def calc_information_gain(impurity, column, branches, databelow, dataabove):
```

```
    if branches == None:
        information_gain = impurity(column) - impurity(databelow) - impurity(dat

    else:
        information_gain = impurity(column) - np.sum(impurity(branches))


    return information_gain


def find_column_split_gains(df, impurity_method):
    if df.shape[1] == 1:
        return df # only one column

    timer_start = time.time()
    gains = []


    # calculate gains for each column (including TransactionID and isFraud!)
    for column in df:
        if column in categorical:

            branches = categorical_split(df, column)
            information_gain = calc_information_gain(impurity_method, column, br

        else:
            best_valsmids = best_split_threshold(df, column, impurity_method)
            splits =  numerical_split(df, column, best_valsmids)
            information_gain = calc_information_gain(impurity_method, column, No

        gains.append(information_gain)

    best_gain = df.columns[np.argmin(gains)]
    print(f"best gain={best_gain}")
    timer_end = time.time()

    print(f"time to calculate {impurity_method.__name__} gain for {len(df)} rows

    return gains

### ##########################################################################


class DecisionNode:
        max_depth = 1
```

```
        def __init__(self):
            self.chance_of_1 = {}     # { value: float }
            self.branches = {}            # { value: None | Node }

        def fit(self, column, y):
            if column.empty: # or any stopping criteria ...
                return None

            for value in column.unique():
                y_value = y[column == value]

                self.chance_of_1[value] = sum(y_value) / len(y_value)

                self.branches[value] = None # guaranteed to be 0, or guaranteed

                if DecisionNode.max_depth > 1:
                    if self.chance_of_1[value] != 0.0 and self.chance_of_1[valu

                        child_node = DecisionNode()
                        child_node.fit(column[column == value], y[column == valu
                        self.branches[value] = child_node

            return self

        def predict(self, value):
            if value not in self.branches:
                return None # value not encountered in training

            chance_of_1 = self.chance_of_1[value]
            child_node = self.branches[value]
            if child_node == None:
                return chance_of_1  # this is a leaf...
            else:
                return chance_of_1 * child_node.predict(value)


    def build_and_predict(df, column):
        exit("not implemented yet")


    # from class ...
    def adjust(n_zeros, n_ones):
        minor = min(n_ones,n_zeros)
        major = max(n_ones,n_zeros)
        imbalance_rtaio = imbalance_ratio(minor,major)
```

```
    weighted_total = (major * imbalance_rtaio) + minor # W_TOTAL = X0*IR + X1

    if weighted_total == 0:
        exit('divide by zero: ir=',imbalance_rtaio,'minor=',minor,' major=',majo

    weighted_major = major * (imbalance_rtaio / weighted_total) # X0_new = X0*IR
    weighted_minor = minor / weighted_total
    return (weighted_minor, weighted_major)
```

Appendix: demo_70%_onetree_nodrop_undersample.py

```
import numpy as np
import pandas as pd

import os
import time

from sklearn.model_selection import train_test_split
from sklearn.utils import resample

### team library ###########################################################
import demo_basics as demo



train_size = 0.99
column = "card1"


### ########################################################################
# load data
df_train = pd.read_csv(os.getcwd() + '/train.csv')
#df_train = demo.remove_iterative(df_train, ["TransactionID"])
#df_train = demo.drop_missing_data(df_train)


accuracy = []
weighted_accuracy = []
found_accuracy = []
```

```
test_size = 1-train_size
print("="*15, "train_size=", int(train_size*len(df_train)), "-", "test_size=", i

print("training size=%.2f" % train_size)
print("test size=%.2f" % test_size)

one_tree = demo.DecisionNode()



X = df_train.iloc[:,:-1]
y = df_train.iloc[:,-1]



# undersample
n_ones = y.sum()
n_zeros = len(y) - n_ones


#noreplace 70%
#replace   70%

X_0, y_0 = resample(
    X[y == 0],
    y[y == 0],
    replace=True,
    n_samples=n_ones,
    random_state=0
)

X = pd.concat([X[y == 1], X_0])
y = pd.concat([y[y == 1], y_0])


X_train, _, y_train, _ = train_test_split(X, y,
                                                    train_size=train_size,
                                                    stratify=y,
                                                    shuffle=True,
                                                    random_state=0)


timer_start = time.time()
one_tree.fit(X_train[column],y_train)
```

```
timer_end = time.time()
print(f"time to fit one-tree node: {timer_end-timer_start}")



df_test = pd.read_csv(os.getcwd() + '/test.csv')
X_test = df_test[column]


timer_start = time.time()
predictions = []
for value in X_test:


    chance_of_1 = one_tree.predict(value)
    if chance_of_1 == None:
        predictions.append(0)     # randint(2)
    elif chance_of_1 < 0.5:
        predictions.append(0)
    else:
        predictions.append(1)

timer_end = time.time()
print(f"time to predict with one-tree node: {timer_end-timer_start}")

start_at = 472433
out_index = range(start_at, start_at+len(df_test))
out_predictions = predictions
out_pd = pd.DataFrame({"TransactionID":out_index, "isFraud":out_predictions})
out_pd = out_pd.set_index("TransactionID", drop=True)
out_pd.to_csv(os.getcwd() + "/out_onetree.csv")
```