Credit Card Fraud Detection with Decision Tree

Northeastern University

CS5100 Final Project

Index

1. **Introduction** -- 3. #

2. **Methodology** -- 5. #

2.1 **Data Analysis**-- 5. #

2.2 **Algorithm** --- 10. #

2.3 **Information Entropy and Information Gain**-- 10. #

2.4 **Code Walkthrough**-- 12. #

3. **Result** --- 15. #

3.1 **Model Validation**-- 15. #

3.2 **Performance Result**-- 15. #

4. **Conclusion**-- 17. #

4.1 **Limitations and Future Improvement** -- 17. #

4.2 **At End**--- 18. #

4. **Reference**-- 19. #

Introduction

Credit card fraud is a pervasive issue that banks face daily, leading to substantial financial losses for both consumers and financial institutions. In 2023 alone, approximately 426,000 cases of credit card fraud were reported to the Federal Trade Commission (FTC), highlighting the urgency and scale of this problem. This project aims to develop a robust algorithm capable of detecting suspicious activities on users' credit cards and immediately freezing them to prevent further unauthorized transactions. This approach not only protects the users' finances but also reduces the administrative burden and financial repercussions for banks.

From personal experiences to theoretical knowledge, our diverse backgrounds as computer science students allow us to tackle this problem from multiple angles. One team member recounted an incident where their card information was stolen, leading to a tedious process of reclaiming funds through multiple forms. Immediate detection and response by the bank could have mitigated this hassle significantly. Another member emphasized the societal impact, noting how credit card fraud induces stress and financial strain on victims.

We propose a machine learning-based solution to classify and predict fraudulent transactions using Decision Trees. This model will evaluate various indicators of fraud, such as significant geographical discrepancies between the cardholder's address and the transaction location, or uncharacteristic purchases that deviate from the user's typical spending patterns. Additional contextual information, such as changes in the cardholder's marital status, could also be considered to refine the predictions.

The accuracy of our fraud detection system is paramount. Not only must it minimize false positives to avoid inconveniencing customers, but it must also reliably detect fraudulent activities to prevent financial losses. By achieving high accuracy, we enhance trust in the banking system and improve customer satisfaction.

Given the challenging job market, especially in the technology sector, engaging in a project with real-world applications such as this not only enhances our skills but also improves our employability. By demonstrating our ability to address significant, current challenges using advanced technologies, we position ourselves as valuable candidates for roles in the ever-evolving field of computer science.

The system will process each transaction in real-time, assessing the probability of fraud. If this probability exceeds a predefined threshold, the transaction will be flagged as suspicious, prompting immediate action from the bank to freeze the card and notify the customer. While no fraud detection system can achieve perfect accuracy, our goal is to significantly reduce the incidence of credit card fraud, thus safeguarding financial assets.

By leveraging our collective expertise and the power of machine learning, we aim to create a fast and efficient fraud detection system that can operate seamlessly within bank data centers, offering a proactive approach to protect consumers and minimize financial losses due to fraudulent activities.

Methodology

Data Analysis

Data acquisition - Our dataset has been sourced from Kaggle. This dataset provides a variety of attributes valuable for comprehensive analysis. It contains 555,719 instances and 22 attributes, a mix of categorical and numerical data types. We chose this dataset because its variety of attributes mimics real-world transaction data with labelled fraud transactions.

Dataset URL: https://www.kaggle.com/datasets/kelvinkelue/credit-card-fraud-prediction/data?select=fraud+test.csv

Data preparation – For the data preparation we handled missing values and encoded the categorical variables. For any cell with missing value, we just simply drop that row. We believe we have enough dataset, and dropping some rows will not damage our dataset. After going through all the features of the dataset we have selected the most relevant and predictive features.

```
# Drop rows where any cell is empty
df_cleaned = df.dropna()
```

Data Analysis – Out of the 22 features we have only selected category, amount, gender, and age (derived from dob). We brainstormed the possible relationship between each feature and the result we tried to predict.

Feature Engineering – We need to do feature selection and feature transformation within our dataset (or data frame).

Feature Selection - First, we pick category, merchant, gender, transaction amount, coordinate of cardholder's address and coordinate of merchant address to get distance, transaction time as features to build our model. We realize we need 10 different columns before feature transformation to start with. The model we tested at first is extremely slow and memory costly. We decided to drop multiple columns to speed up our model. In fact, choosing too many features can lead to other common errors in machine learning, which is overfitting. After we give a couple tries, we only pick four features left, gender, category, transaction amount, and date of birth of cardholder.

Feature transformation – The feature gender and category is not numerical initially, we need to change them into number so our algorithm can understand. The way we do it is using pd.get_dummies() as one-hot encoding. We replace these two columns with new columns such as gender_M or gender_F. Those new columns only contain 1 or 0. The code below will achieve it:

```
df_encoded = pd.get_dummies(df, columns=['category', 'gender'], dtype=int, drop_first=True)
```

The other thing is we switch the date of birth to cardholders' age with:

```
df['dob'] = pd.to_datetime(df['dob'], dayfirst=True)
current_date = datetime.now()
df['Age'] = (current_date - df['dob']).dt.days // 365
```

Though date is can be converted to numerical value, the number we converted to will be extremely large. Since we decide not to normalize our dataset, these large numbers can affect the predict result a lot. We believe simply converting into people's age can be a good choice.

The reason we decide not to normalize our data such as age and transaction amount can be more meaningful for our model. More money spends usually happen during scam, and elder people can be easily tricked at the most of time. We want the larger number to have more weigh.

Once our features is ready, we use a powerful tool in python called matplotlib.pyplot to draw some diagram to analysis our data. We want to see the distribution of each feature. The 3 figures below will display the basic impression about how our dataset is:

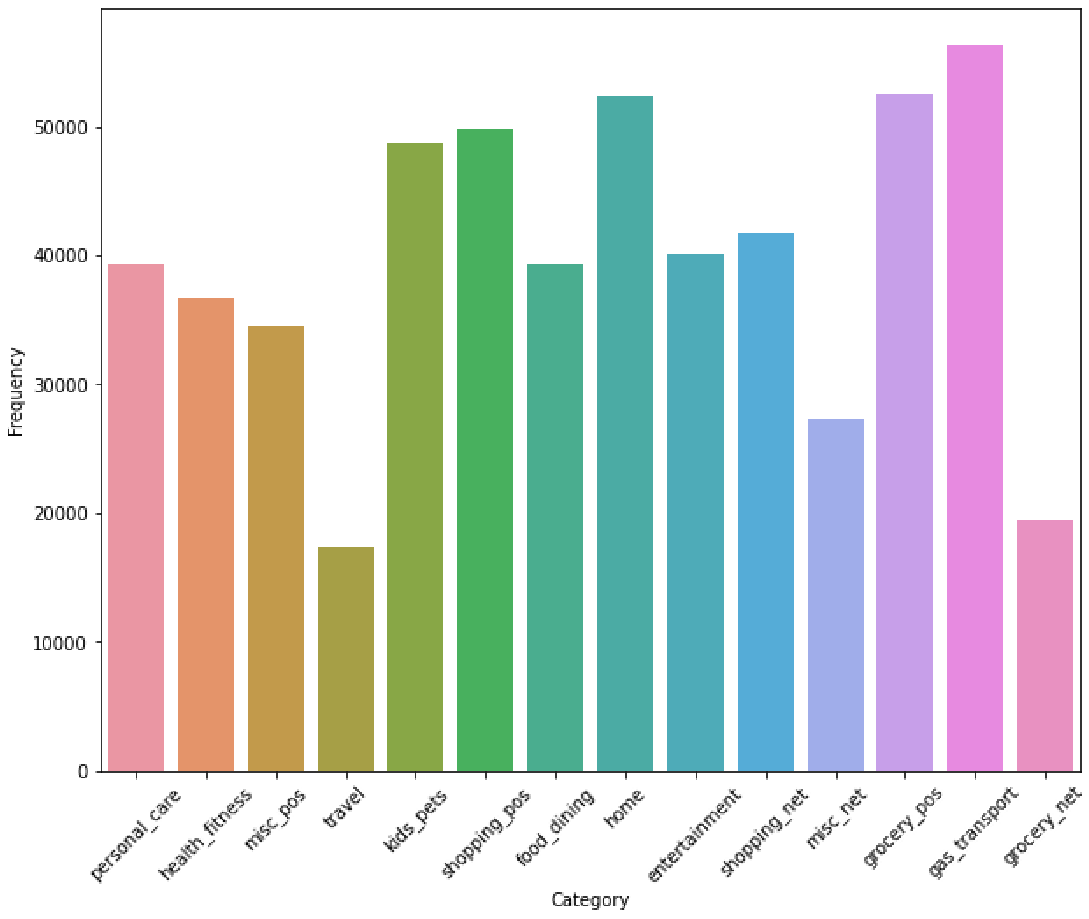

Figure 2-1 Category Distribution

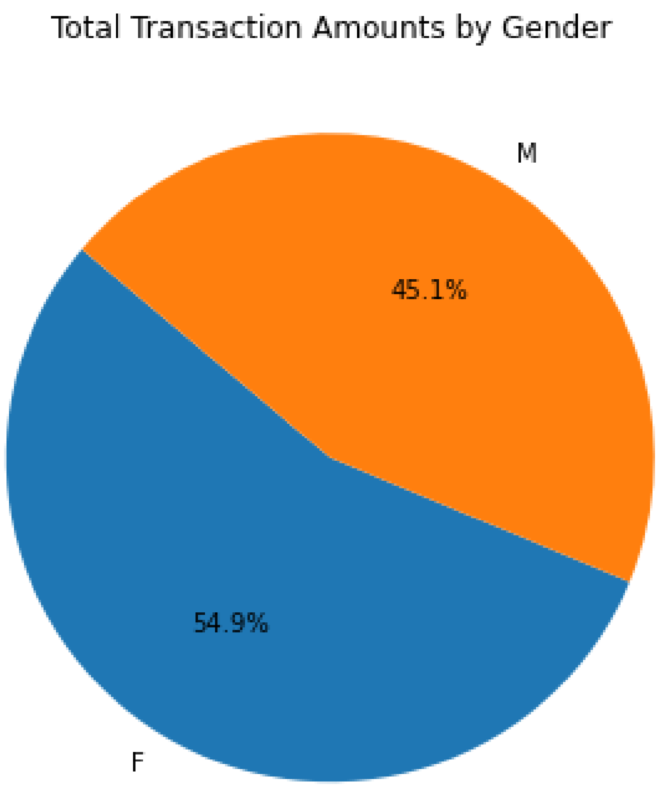

Figure 2-2 Distribution of Gender

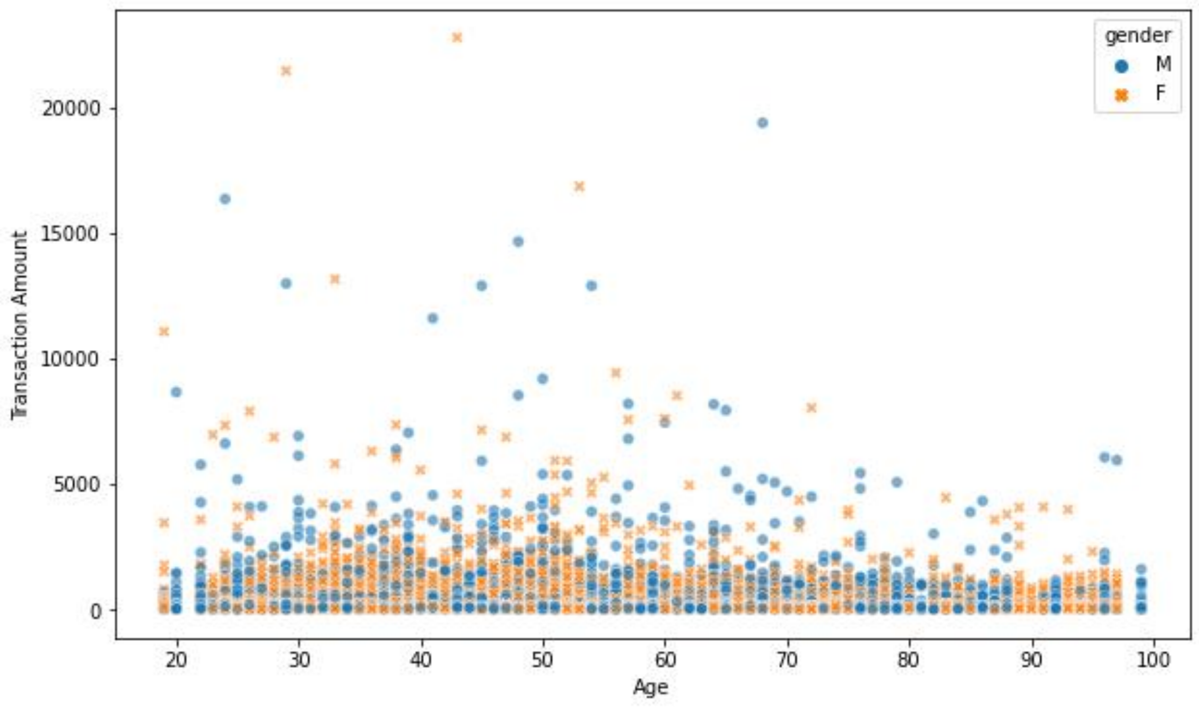

Figure 2-3 The Relationship between Age and Money Spent

From figure 2-3, we notice some spending are extremely large and transaction happens from someone is too old (Above 90 years old). In the first, we thought we should remove these abnormal feature value since our algorithm may be heavily impacted by it. However, the topic we are dealing with may need this absurd value after all, since this value can be fraud the most of time. In general, we do not believe someone will swiping their credit card with their age of 98. At this point, our data is ready, and it is time to show the basic idea of our algorithm—Decision Tree Classifier.

Algorithm

Each algorithm is built from knowledge of math. For decision tree, we discover there are 4 different types of people used. They are ID3, C4.5, C5.0, and CART. The first import term among them is information gain. In each decision node, it hope it can have more information after it split itself into two new child nodes. For the first 3 types, the way to achieve it is calculating a new term—information gain. To make each split the best split, algorithm is hoping for get the most information gain out of it. The idea is simple, the decision node will start split its current dataset by randomly select feature and evaluates these selected features to find the best points to split the data at current node. It will scan through the value of each selected feature, and calculating the information gain to determine how well a split on each unique value would divide the data into homogeneous subsets. In order to understand the term information gain, we need to learn what is information entropy first.

Information Entropy and Information Gain

People use information entropy to determine how chaos the dataset is. If a dataset contains various classes with different features, we know that the element from that dataset is hard to distinguish which class it belongs to. In this case, the entropy should be high. Higher entropy represents the data are spread out as much as possible, while low entropy means that the data are nearly are concentrated. For our decision tree algorithm, higher entropy stands for unpredictable. Entropy will be directly related with information gain, which is the other important key term in this algorithm.

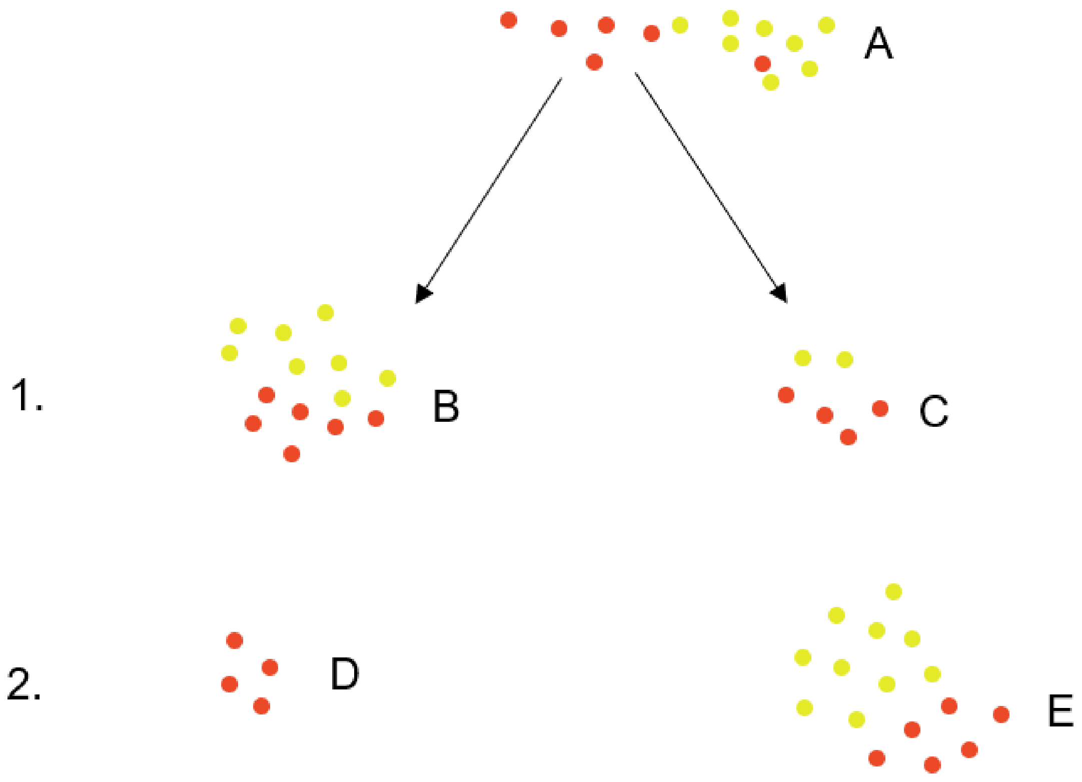

Figure 3-1

In figure 3-1, five different group above contain 2(or 1) classes of elements, we can calculate the entropy from each group. We have 10 red elements and 10 yellow elements in total in each layer. Here, we use the following formula:

$$Entropy = \Sigma - p_i \log p_i$$

p_i is the probability of class i

A little fun fact, the unit of entropy is defined by the base of log. If the base of log is 2, the unit is bits; if the base of log is e, the unit is nats; if the base of log is 10, the unit is bans.

With the formula, we can get entropy from our 5 group:

A: $-0.5\log(0.5) - 0.5\log(0.5) = 1$

B: $-\frac{6}{14}\log\left(\frac{6}{14}\right) - \frac{8}{14}\log\left(\frac{8}{14}\right) = 0.99$

C: $-\frac{2}{6}\log\left(\frac{2}{6}\right) - \frac{4}{6}\log\left(\frac{4}{6}\right) = 0.91$

D: $-1\log(1) - 0\log(0) = 0$

E: $-\frac{10}{16}\log\left(\frac{10}{16}\right)-\frac{6}{16}\log\left(\frac{6}{16}\right)=0.95$

We can see in group D, there is only 1 color. This group is deterministic, so the entropy here is 0. Notable, this group only contain single elements, which should be defined as the leaf node of our decision tree, since we are 100% sure if an unknown element ends up here, it belongs to this class. Meanwhile, for other 4 group with relatively high entropy, we will not determine the class which element belongs to.

With the entropy, the decision node in our decision tree can decide how to split the dataset. We need to use the other measurement: Information Gain (IG).

Similar to entropy, we have the formula for information gain as well. From the same figure 3-1, there is 2 different ways to split our dataset into to sub dataset, method 1 and method 2. Our algorithm has to know which method we should pick. With the entropy we calculated, we can use the formula below to calculate information gain, which is the deterministic factor about how each dataset (or sub-dataset) is spited.

$$Information\ Gain = E - \Sigma \mathrm{w_i} E(child_i)$$

$\mathrm{w_i}$ represent the weight of each split i (or sub dataset i) compared to their parent

With this formula, we can calculate the information again of each split. Our algorithm will choose the best one and repeat this process until we reach the leaf node.

Let’s apply the formula to our split:

1. $IG_1 = 1 - \frac{14}{20} * .99 - \frac{6}{20} * .91 = 0.034$
2. $IG_2 = 1 - \frac{4}{20} * 0 - \frac{16}{20} * .95 = 0.24$

The 2nd split has higher information gain. That the split we are looking for. The number is matching with our intuition, since a leaf node with only one class element is being created after this split.

With entropy and information gain, we can split our tree further. The algorithm is well known as ID3 algorithm in decision tree.

Code Walkthrough

For the decision tree class:

The class constructor takes three parameters:

1. Min_samples_split: this is the minimum number for a split. In other words, if the number of samples is less that this number, we stop growing the tree to avoid overfiiting.
2. N_features: this is the number of features we will use to build the tree.
3. Max_depth: this is the maximum depth of our tree. It is used to avoid overfitting as well.

The method **fit()** initializes the number of features from the data and creates the root node.

The method **create_tree()** builds the tree by recursively calling itself: first, it checks the conditions for stopping the recursion, then randomly selects features to do the split by calling the best_split() method.

```
feat_index = np.random.choice(n_feats, self.n_features, replace=False)
best_feature, best_threshold = self.best_split(X, y, feat_index)
```

After we do the split of the data, we can build the left and right sub-trees by calling the create_tree() on the split data separately. For the case of leaf nodes, we do not need to further split. Instead, we call the **most_common_label()** method to find which kind of data set is the majority in this leaf node and set this leaf node correspondingly. In our case, this is simply “is fraud” or “not fraud”.

The method **best_split()** takes a list of features that we are interested in. For each feature, we try all possible splits and compute the corresponding information gains. We can thus find the best feature to do the split and take the best split on it.

The methods **information_gain()** and **entropy()** compute the information gain based on the entropy by the formula we talked about in the entropy and information gain section, which means the information gain is the difference between the entropy before the split and the entropy after the split. This is the heart of algorithm, and we will demonstrate how we transfer the math formula into code:

```
def entropy(self, y):
        hist = np.bincount(y)
        ps = hist / len(y)
        return -np.sum([p * np.log(p) for p in ps if p>0])
```

```
def information_gain(self, y, X_column, threshold):
        parent_entropy = self.entropy(y)
        left_index, right_index = self.split(X_column, threshold)
        if len(left_index) == 0 or len(right_index) == 0:
            return 0
        n = len(y)
        n_l, n_r = len(left_index), len(right_index)
        e_l, e_r = self.entropy(y[left_index]), self.entropy(y[right_index])
        child_entropy = (n_l/n) * e_l + (n_r/n) * e_r
```

The methods **predict()** and **traverse_tree()** are used to predict whether a new transaction is a fraud: given the new date point x, We traverse the tree by comparing the values of features of x and the thresholds we set at each tree node. This is how we get our result with the tree we just created.

After that, our classifier is ready to work, we just need to pass the data we prepared in and train it.

Model Building – Running our algorithm on the dataset we achieved weirdly high accuracy. This high accuracy might suggest overfitting. We then noticed the is_fraud column which values we had to predict. There are significantly less fraud transactions (only 2145 out of 555720) and this severe imbalance is the reason for getting such high accuracy. In fraud detection, high accuracy does not necessarily indicate great model performance as the data is heavily skewed towards non-fraudulent transactions. In real-world scenarios, fraudulent transactions are indeed less compared to legitimate ones. Therefore, relying only on accuracy as a performance metric can be misleading. The way we try to solve this unbalanced problem is using Synthetic Minority Over-sampling Technique (SMOTE). There is a package about how to deal with imbalance data in python called imblearn.over_sampling. After we split our dataset into 80% training parts and 20% testing parts, we use SMOTE to modify our training parts. This process should give us a more accurate model.

```
smote = SMOTE(sampling_strategy='auto') #To Balance the data as output is imbalance in the dataset.
x_train_smoted, y_train_smoted = smote.fit_resample(x_train, y_train)
```

Result

Model Validation

We decide to use the built-in decision tree from sklearn as well to compare the result between our algorithm and their algorithm. We use accuracy_score function from sklearn package to check the result we got. This is what we have for a random try:

```
accuracy for self-built tree:  0.9925
accuracy for sklearn built-in tree:  0.997
```

We are so happy at first since our algorithm is so close with the one sklearn built. We decide to evaluate the performance more using other metrics. The performance of the tree was evaluated using different metrics like accuracy, precision, recall, and classification report. These are very crucial metrics as the false positives (legitimate transactions flagged as fraudulent) and false negatives (not detecting fraudulent transactions) are very important in fraud detection. Precision measures how accurate the positive predictions made by the model and recall assesses that as many fraud cases as possible are detected.

Additionally, we have compared the accuracy, precision and recall of the training and test datasets to identify any signs of overfitting and see how well the model generalizes to new data. For the training set these metrics evaluate our model's ability to learn the underlying patterns and for the test set these metrics provide insight into our model's predictive accuracy.

Performance Result

```
Training Accuracy: 0.9985597996242955
Testing Accuracy: 0.9925
Training Precision: 0.9972520609542843
Testing Precision: 0.16666666666666666
Training Recall: 0.9998747651847214
Testing Recall: 0.2857142857142857
Classification report:
               precision    recall  f1-score   support
           0       1.00      0.99      1.00      1993
           1       0.17      0.29      0.21         7
    accuracy                           0.99      2000
   macro avg       0.58      0.64      0.60      2000
weighted avg       0.99      0.99      0.99      2000
```

We realize our model is trained badly based on the precision and recall score for 1 (is_fraud =

True) we get. In fact, the lower precision score for 1 is fine in our topic. If we are the bank, we can consider some non-fraud into fraud and block that transaction. This situation common happen in our life as a precautionary measure. However, the lower score for recall part is unacceptable. It indicates that we define many frauds into not_frauds. This is critical mistake in our topic. The higher accuracy happen due to other data is imbalanced. We have too many non-frauds in our both training and test part. Due to our dataset is super larger, the little number of mis-prediction for 1 does not affect the accuracy a lot with this large sample size. Putting a small stone into desert will not make people forget this is desert. However, the lower score in recall shows this is a serious problem.

The other thing worth to mention is that our own algorithm is running really low. Though we tried to change the threshold to improve the performance, the result we get is unacceptable. We have 555720 samples in total, but we can run only 100000 sample at most for my own personal computer.

```
if random.random() > 0.5:
    df = df.head(100000)
else:
    df = df.tail(100000)
```

We only use these lines to test the first or last 100000 sample. In comparison, the built-in model in sklearn run fast. It can finish within a minute.

Conclusion

Limitations and Future Improvement

In our decision tree, we only utilized four categorical features: age, gender, transaction type, and amount for simplicity. However, there are several more features that could potentially enhance the model's predictive power. For instance, incorporating the distance between the cardholder's location and the merchant's location computed by their latitude and longitude, as well as the time of the transaction, could provide valuable insights. Integrating these additional features might lead to a more robust and accurate model.

The other thing is the running speed of our own model as what we discussed. It is slow when the sample size is larger. We believe this happen because our time complexity is pretty bad. This may happen because we used too many nested loops with recursion when we create our tree. How to make our tree run more efficiently should be one of our goals.

Our dataset may be too real for our model. This imbalance dataset definitely decreases our reliability of our result. The one solution should be simple, we just need to find some balanced dataset to train our model, though it is not real. We can apply our model into the real data after ward once our model performs well. The other way is we need to create more duplicated sample in our dataset and make each class relative balanced. Only 2145 out of 555720 samples belong to one class does not work when it only 0.3% of our total samples. However, we can duplicate this 2145, and make this class hold for 10% among our datasets. That may lead to a better result.

When determining the best split in our decision tree, we employed the np.unique() method to generate a list of all values to be tested. While this approach is suitable for categorical features or features with a limited number of values (such as ages in our case), it may not be ideal for features with continuous values. In such cases, using a regression decision tree could be more effective in handling continuous real values, thus improving the model's ability to generalize.

The decision tree algorithm is inherently susceptible to overfitting because it does not inherently limit its own depth, which can lead to increased accuracy but at the cost of model complexity and generalizability. To address this issue, it would be beneficial to employ decision tree pruning techniques. One approach is to use "pre-pruning," where a maximum depth (max_depth) parameter is set to limit the depth of the decision tree during its construction. We used this technique pre-pruning in our tree. However, the parameter should be optimized in the future. We should consider more situations and set the parameter related to our dataset. Additionally, "post-pruning" can be applied after the tree is built by modifying its structure to improve its ability to generalize to unseen data. Implementing these pruning techniques can help reduce overfitting and enhance the predictive accuracy of the decision tree model.

The last, a single decision tree seems unstable for our prediction. One tree does not finish the job well, what about we try to create a forest. Random Forest may perform better in our topic. Switching other machine learning technique is always worth a try.

At End

In conclusion, credit card fraud remains a significant threat, impacting millions of individuals and businesses worldwide. While our project to implement a Decision Tree algorithm for detecting credit card fraud did not meet all our initial expectations, it provided us with significant insights and a deep understanding of the complexities involved in fraud detection. Decision Trees, while advantageous for their simplicity and efficiency in handling large datasets, also presented challenges such as susceptibility to overfitting and sensitivity to noisy data. These challenges underscored the importance of choosing the right model and tuning it carefully.

Despite not achieving the level of success we anticipated, our proactive approach significantly contributed to our learning and professional development. We gained invaluable experience in applying machine learning techniques to real-world problems, understanding the limitations of our tools, and exploring potential improvements. This project was not just about achieving perfect results but was a profound commitment to learning and growth in the field of financial security.

Moving forward, we are equipped with a better understanding of what is required to develop more effective fraud detection systems. We remain committed to refining our approach and continuing our research, driven by our ultimate goal of contributing to safer financial transactions and a more secure digital environment.

Reference

“Decision Tree Classification in Python (from Scratch!).” *YouTube*, YouTube, 21 Jan. 2021, www.youtube.com/watch?v=sgQAhG5Q7iY.

“How to Implement Decision Trees from Scratch with Python.” *YouTube*, YouTube, 15 Sept. 2022, www.youtube.com/watch?v=NxEHSAfFlK8&t=152s.

Elmaravilla. “Fraud Detection Using DTC 99% Accuracy.” *Kaggle*, Kaggle, 29 Mar. 2024, www.kaggle.com/code/elmaravilla/fraud-detection-using-dtc-99-accuracy.

Kelue, Kelvin. “Credit Card Fraud Prediction.” *Kaggle*, 11 Mar. 2024, www.kaggle.com/datasets/kelvinkelue/credit-card-fraud-prediction/data.

Navlani, Avinash. “Python Decision Tree Classification Tutorial: Scikit-Learn Decisiontreeclassifier.” *DataCamp*, DataCamp, 23 Feb. 2023, www.datacamp.com/tutorial/decision-tree-classification-python.

Raj, Ashwin. “An Exhaustive Guide to Decision Tree Classification in Python 3.X.” *Medium*, Towards Data Science, 11 Dec. 2021, towardsdatascience.com/an-exhaustive-guide-to-classification-using-decision-trees-8d472e77223f.

Saini, Anshul. “What Is Decision Tree? [A Step-by-Step Guide].” *Analytics Vidhya*, 18 Apr. 2024, www.analyticsvidhya.com/blog/2021/08/decision-tree-algorithm/#:~:text=A%20decision%20tree%20is%20a,easy%2Dto%2Dunderstand%20models.