PROJECT 2 – CLASSIFICATION

In the project we will go through the data science process with the goal of classifying property loans.

Jesus Barrera Mejia

ITCS 3162 002

Professor Alieen

2/23/2024

The Problem/Question & The Data

My model will attempt to decide if someone is approved for a property loan based on 11 features.

The 11 features are:

1. **Gender:** Male or Female
2. **Married:** Married or Not Married
3. **Dependents:** 1, 2, or 3+
4. **Education:** Graduate or Undergraduate
5. **Employment:** Self-employed or not self-employed
6. **Income:** In Dollars
7. **Coapplicent Income:** In Dollars
8. **Loan Amount:** In Thousands
9. **Loan Terms:** In months
10. **Credit History:** 1 if the credit history meets the requirements for the loan
11. **Property Area:** Semi-Urban, Urban, or Rural

My data set was retrieved from Kaggle - (https://www.kaggle.com/datasets/bhavikjikadara/loan-status-prediction/data). One problem I encountered with this dataset is that the author did not provide information on where he pulled all the data. The author is from India, so the data may be from India. Secondly, there are no dates provided in the data. The problem with this is that different loans may be from different dates so we will not be able to gain insight into the influence of the year on loan approvals. Additionally, no dates will make it harder to tell where the data set is from. If we had dates we could have been able to look at the average salary and gotten an idea if it's from the U.S. or not based on average salaries for different years in the U.S. Lastly, the data set is ***unbalanced*** meaning that it has more entries from one classification than the other. This is made worse by the fact that the data set is small with only 381 entries before preprocessing.

Pre-processing

Data Types

- When we use the dtypes function on the data, it reveals that 6/11 of our features are categorical. These are the features we will need to *one-hot encode* later.

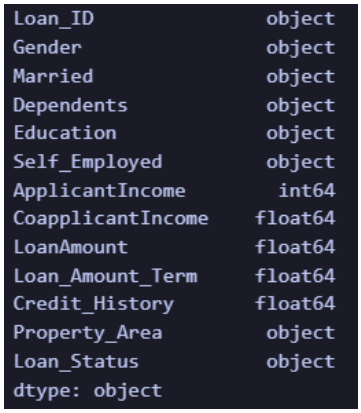

Figure 1

Null Values

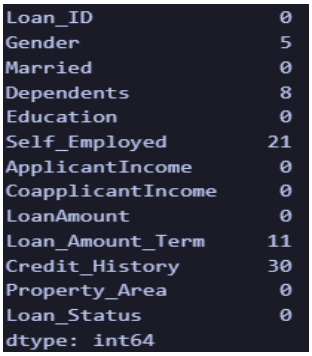

Figure 2

- My dataset had null values for 5 values. **Dependents**, **Gender, Self_employed, Loan_Amount_Term, & Credit_History**.
- For gender and credit history, I did not drop the nulls. Instead, I replaced the nulls with a "Not_disclosed" value. My thought process was that these were values that people in the real

world sometimes do not provide, so it would make sense for some of these to be null. After this process, we have Dependents, self_employed, and Loan_amount_term to handle.

- For the last three columns, I dropped the values. I couldn't find a good reason why they would be null. Self_employed specifically had me a little confused because I was not sure if 0 included unemployed people or not, for simplicity, I will assume that it does.

```
#Replacing Null Values for Gender
data["Gender"].fillna(value = "Not_disclosed", inplace=True)

#Replacing Null Values for Credit History
data["Credit_History"].fillna(value = "Not_disclosed", inplace=True)
```

```
#To prevent bias we will drop the remaining nulls
data = data.dropna()
```

One-Hot Encoding

```
data = pd.get_dummies(data, columns=['Gender', "Dependents", 'Credit_History', 'Property_Area'])
✓ 0.0s
```

```
#Replace values with 1 & 0
data["Loan_Status"] = data['Loan_Status'].map({'Y': 1, 'N': 0}) #Forgot to do this on, this is the actual value
data["Married"] = data['Married'].map({'Yes': 1, 'No': 0})
data["Self_Employed"] = data['Self_Employed'].map({'Yes': 1, 'No': 0})
data["Education"] = data['Education'].map({'Graduate': 1, 'Not Graduate': 0})
```

- For Gender, Dependents, Credits History, and Property Area I used the get_dummies function to create the one-hot encoding. The reason was that these were the columns with multiple bins, so as such they required more than one column. For Loan Status, Married, Self-employed, and Education we just did the one hot encoding within the column itself since they used binary classification.

Formatting

I'm just going to change the income columns to yearly salary

```
dataForModel["ApplicantIncome"] = dataForModel["ApplicantIncome"] * 12
dataForModel["CoapplicantIncome"] = dataForModel["CoapplicantIncome"] * 12
✓ 0.0s
```

- For my last step in preprocessing, I formatted the Applicant Income & Coapplicent Income into yearly amounts. I don't receive any information about what format they are in from the data sets source, but when transformed into yearly amounts they make sense. The only odd thing was that the co-applicant's income was much lower than the applicant's income.

Visualizations

Heatmap

- For decision trees, multicollinearity is not a problem. What I'm looking for is for high correlation between features and loan status (The value we are trying to predict). The feature that stands out is credit history. In our dataset, credit history is 1 if the applicant's credit history meets the requirements for the loan. The correlation matrix tells us that not having the required credit history has a negative correlation with loan status, meaning it can predict if you won't be approved. It's the same thing with having the required credit history, the only difference is that it's a slightly lower positive correlation.

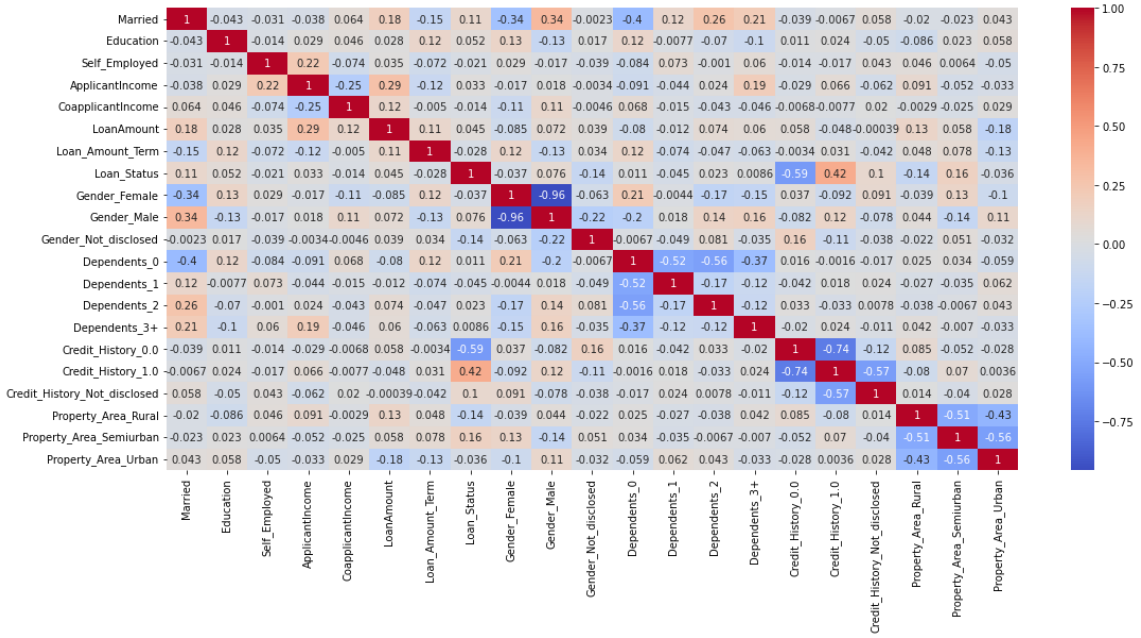

Figure 3

Statistics

- Before creating our models, I wanted to point out some important statistics within our data.

- **Applicant Income** has an average yearly income of $42,850.
- The Average **Loan Amount** is for $104,000
- The average **Loan Term** is for 30 years
- **Approved Loans** make up 71% of the data

Modeling

Splitting & Weighting the Data

```
#Creating Variables
X = dataForModel.drop(columns = ["Loan_Status"])
y = dataForModel["Loan_Status"]
✓ 0.0s
```

```
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size = 0.2, stratify=y)
```

```
from sklearn.utils import class_weight
weights = class_weight.compute_sample_weight('balanced', y_train)
```

- For my models, I will use the DecisionTreeClassifier provided by scikit-learn.
- We start by separating the data into our features and our classifications (X & Y, respectively)
- Second, we split our data using 20% of our data for testing the model.
- Lastly, we add weights to our samples. The reason we do this is to avoid bias because our data is unbalanced.

The How & Why of The Decision Tree

- The decision tree works by calculating a ***Gini value*** for each feature. The ***Gini value*** is a measure of impurity for our features; the higher it is, the less powerful the feature is for aiding in prediction. Using the Gini value, we want to find the features that are associated with our classes. The reason we use the decision tree is because of its interpretability. Using the plot_tree function we can visualize our tree and understand which features are important to classifying

our loans. Additionally, decision trees are easy to understand and easy to use, so it works great for beginners.

- The disadvantage to using the decision tree is that it may overfit the data, meaning that it is only good at predicting data from our dataset. To get around this problem we split our data, but still, this is done at random and slightly altering the data input could result in a very different tree. Also, decision trees are good for the quantity of data we are using, but as data sets get larger, they may not be as efficient.

Decision Tree 1 Evaluation

```
plot_confusion_matrix(first_model, X_test, y_test, values_format='d', display_labels= ["Loan Approved", "Loan Not Approved"])

tn, fp, fn, tp = confusion_matrix(y_test, predictions1).ravel()
precision_score = tp / (tp + fp)
recall_score = tp / (tp + fn)
print(f"Precision: {precision_score} \nRecall: {recall_score}")
✓ 0.1s

Precision: 0.8269230769230769
Recall: 0.8775510204081632
```

- For our first model, we get a general idea of how well the decision tree will be without any fine-tuning. We will be measuring our decision tree using a **Confusion Matrix.** The value we are most interested in is ***Precision***. Precision is a of measure how many of our positive predictions were true positives, which is most important for loans: i.e. we don't want to approve someone who

shouldn't have been approved. 83% is not bad, but this tree is biased since it does not use any weights. Additionally, 83% is probably a result of overfitting. Let's try to see if adding sample weights and changing some parameters provides better results.

Decision Tree 2 Evaluation

- For our next iteration of our decision tree let's find the optimal number of leaves to prevent overfitting. To do this we create a function that measures our precision and recalls at different max-leaf numbers.

```
def get_confusion_matrix(max_leaf_nodes, train_X, val_X, train_y, val_y):
    model = DecisionTreeClassifier(max_leaf_nodes=max_leaf_nodes, random_state=0)
    model.fit(train_X, train_y, sample_weight=weights)
    preds_val = model.predict(val_X)
    tn, fp, fn, tp = confusion_matrix(val_y, preds_val).ravel()
    precision_score = tp / (tp + fp)
    recall_score = tp / (tp + fn)
    print(f"Precision: {precision_score} \nRecall: {recall_score} ")
```

```
    tn, fp, fn, tp = confusion_matrix(y_test, second_model_predictions).ravel()
    precision_score = tp / (tp + fp)
    recall_score = tp / (tp + fn)
    print(f"Precision: {precision_score} \nRecall: {recall_score}")
 ✓ 0.0s

Precision: 0.8837209302325582
Recall: 0.7755102040816326
```

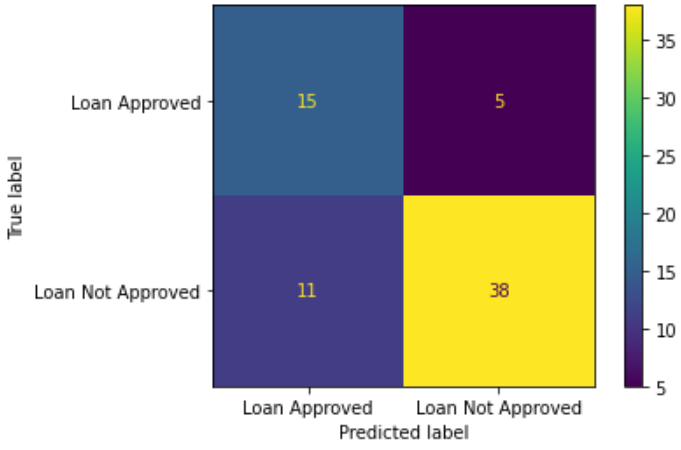

Figure 4

- After running our function, we found that the optimal number of leaves is somewhere around 55 so we used that. This surprised me because I thought a higher leaf number would lead to overfitting, but it's important to note that our recall is lower now meaning our model is not approving a lot of loans that should be approved, but when we look at the confusion matrix (figure 4) we realize that true positives and false negatives make up a very small amount of the test data which could be leading to these results.

Storytelling

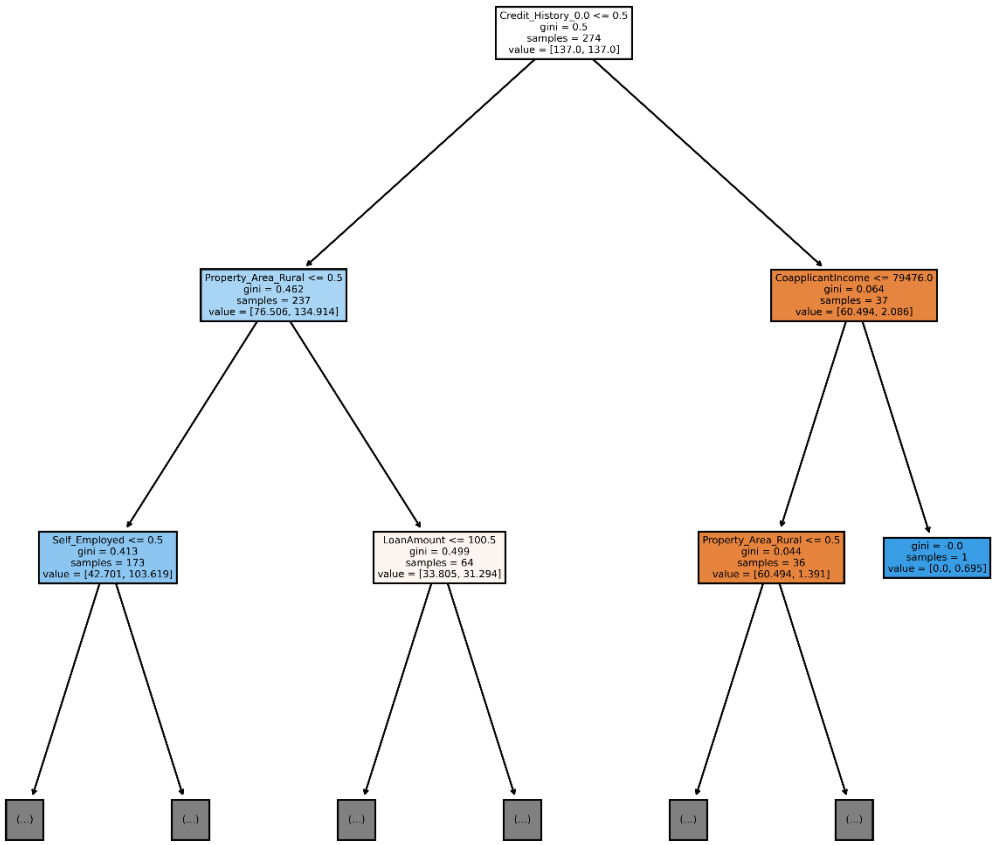

Figure 5

Figure 5 shows an illustration of our second decision tree. For readability, I limited the visible depth to 2. Our most important feature is credit history – which was also the most correlated according to our heat map. If someone meets the credit history, the next purest feature is where the area for the property is. If the property is not in a rural area, then continue with the idea that we will approve the loan. Otherwise, if it is in a rural area then we look at how much the loan is for and the applicant's income. For people who do not meet the credit history, the next purest feature is the co-applicant's income and then similarly we look to see if they are buying a property in a rural area, and this eliminates half of our samples for this branch in the tree and classifies them as not approved.

Overall, our model provides useful insight. The 77% recall rate is concerning, but the 88% precision is promising. We learn that one of the most important features is meeting the credit requirements for the loan and after it's where the property is and how much the loan is for.

Impact

A significant concern for my project is that it could wrongly approve loans that should not be approved. This concern is worsened by the fact that my dataset was unbalanced and had fewer non-approved loans. The positive impact is that it can help someone understand what is important for getting a property loan approved. The model reveals how credit history and where the property can make the difference in being approved for the loan. However, it is important to note that this could just be a correlation made by the model and it could not reflect reality. Regardless, it does provide insight into the loan approval process and could benefit someone looking to understand property loans.

Refrences & Code for Figures

Kaggle Data Set Link:

- https://www.kaggle.com/datasets/bhavikjikadara/loan-status-prediction/data

Figure 2

```
data.isnull().sum()
```
✓ 0.0s

Figure 3

```
#Heatmap visualization
corr = dataForModel.corr()
plt.figure(figsize = (18,8))
sns.heatmap(corr, annot = True, cmap='coolwarm')
```
✓ 2.0s

Figure 4

```
plot_confusion_matrix(second_model, X_test, y_test, values_format='d', display_labels= ["Loan Approved", "Loan Not Approved"])
```
✓ 0.2s

Figure 5

```
from sklearn import tree
fig, axes = plt.subplots(nrows = 1, ncols = 1, figsize = (10,10), dpi=600)
tree.plot_tree(second_model, feature_names = X.columns, filled=True, max_depth=2)
plt.show()
```