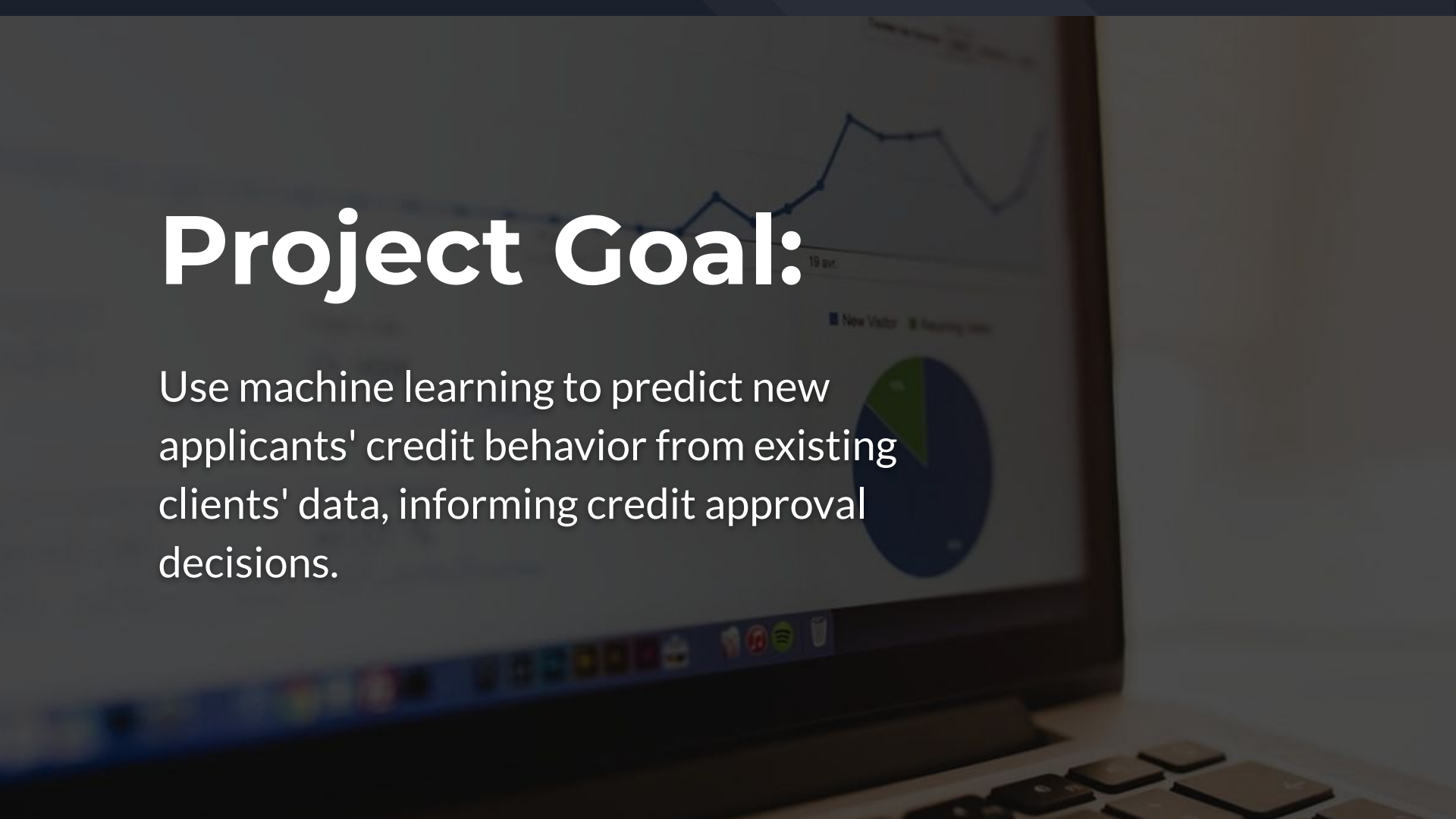
Making Credit Application Decisions With Machine Learning

Jordan Yuhasz



Project Goal:

Use machine learning to predict new applicants' credit behavior from existing clients' data, informing credit approval decisions.

The approach

- **Data Integration:** Merged application and credit records to create a comprehensive dataset.
- **Data Filtering:** Focused on clients with certain payment statuses and a sufficient history to ensure meaningful predictions.
- **Feature Engineering:** Calculated the frequency of payment statuses for each client and used these frequencies as features.
- **Visualization:** Employed histograms and hexbin plots to visualize the distribution of payment frequencies and their relationship with the duration of credit history.
- **Machine Learning Pipeline:** Developed a pipeline incorporating preprocessing (scaling for numerical features and one-hot encoding for categorical features) and a Random Forest Classifier to predict credit decision. Adjusted the decision threshold to optimize model performance.

Challenges

- **Imbalanced Data:** The skewed distribution of payment statuses required careful handling to avoid biased predictions. The class_weight parameter and decision threshold adjustment were strategies used to address this.
- **Feature Selection:** Determining the most predictive features and the appropriate time frame for analyzing payment history required iterative experimentation.
- **Model Tuning**: Confusion from the confusion matrix made things confusing. Understanding this and how to manipulate the values to get the desired results took great effort and time.
- **No Data Regarding Profitability**: I was forced to make filtering decisions based on domain knowledge / personal preference, since there was no data on profitability per demographic. Where I would have rather let profits make these decisions for me.

Results

- **Predictive Performance:** The model demonstrated the ability to predict credit risk with adjustable accuracy, precision, recall, and F1 scores, showing a tangible outcome from the hypothesis.
- **Insights into Payment Behavior:** The analysis revealed significant insights, such as the average frequency of Consistent Credit payment status among certain users and the relationship between delinquent payment appearances and account duration.
- **Model Optimization:** Adjusting the decision threshold and incorporating a modified delinquency filter led to improved model performance, highlighting the importance of fine-tuning in machine learning projects.

The Data

Personal information and data submitted by credit card applicants

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 438557 entries, 0 to 438556
Data columns (total 18 columns):
 #   Column               Non-Null Count   Dtype
---  ------               --------------   -----
 0   ID                   438557 non-null  int64
 1   CODE_GENDER          438557 non-null  object
 2   FLAG_OWN_CAR         438557 non-null  object
 3   FLAG_OWN_REALTY      438557 non-null  object
 4   CNT_CHILDREN         438557 non-null  int64
 5   AMT_INCOME_TOTAL     438557 non-null  float64
 6   NAME_INCOME_TYPE     438557 non-null  object
 7   NAME_EDUCATION_TYPE  438557 non-null  object
 8   NAME_FAMILY_STATUS   438557 non-null  object
 9   NAME_HOUSING_TYPE    438557 non-null  object
 10  DAYS_BIRTH           438557 non-null  int64
 11  DAYS_EMPLOYED        438557 non-null  int64
 12  FLAG_MOBIL           438557 non-null  int64
 13  FLAG_WORK_PHONE      438557 non-null  int64
 14  FLAG_PHONE           438557 non-null  int64
 15  FLAG_EMAIL           438557 non-null  int64
 16  OCCUPATION_TYPE      304354 non-null  object
 17  CNT_FAM_MEMBERS      438557 non-null  float64
dtypes: float64(2), int64(8), object(8)
```

STATUS:

0: 1-29 days past due

1: 30-59 days past due

2: 60-89 days overdue

3: 90-119 days overdue

4: 120-149 days overdue

5: Overdue or bad debts, write-

than 150 days

C: paid off that month

X: No loan for the month

```
STATUS
C    329536
0    290654
X    145950
1      8747
5      1527
2       801
3       286
4       214
Name: count, dtype: int64
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1048575 entries, 0 to 1048574
Data columns (total 3 columns):
 #   Column          Non-Null Count    Dtype
---  ------          --------------    -----
 0   ID              1048575 non-null  int64
 1   MONTHS_BALANCE  1048575 non-null  int64
 2   STATUS          1048575 non-null  object
dtypes: int64(2), object(1)
```

Data Filtering and Exploratory Data Analysis

*Working on the merged df

```
# Filter DataFrame for clients with only 'C' and 'X' and '0' in their STATUS
cx0_clients = df[df['STATUS'].isin(['C', 'X', '0'])]
```

After many iterations, I decided to allow all customers with C, X, and 0 status in the target pool. However, went on to further filter the 0 class.

Current Targets - 766,140

```
STATUS
C    329536
0    290654
X    145950
Name: count, dtype: int64
```

Allowing all users with the 0 status seemed risky, since one user with this status could have only 1 late payment, with a perfect history for all other months. While another may consistently be late as frequent as every month. In my opinion, the first user should not be classified the same as the second user.

Data Filtering and Exploratory Data Analysis

*Building on the previous slide

Acting on the previous consideration, I investigate the users who have at least one late payment.

```
# Step 1: Filter users with at least one '0' appearance
users_with_zero = df[df['STATUS'] == '0']['ID'].unique()

# Step 2 & 3: Calculate the number of '0' appearances and total months recorded for each user
zero_counts_per_user = df[df['ID'].isin(users_with_zero)].groupby('ID')['STATUS'].apply(lambda x: (x == '0').sum())
total_months_per_user = df[df['ID'].isin(users_with_zero)].groupby('ID')['MONTHS_BALANCE'].nunique()

# Step 4: Calculate average number of '0' appearances per 12 months
average_zeros_per_12_months = (zero_counts_per_user / total_months_per_user * 12).mean()
average_zeros_per_12_months_half = average_zeros_per_12_months / 2
average_zeros_per_12_months_half_half = average_zeros_per_12_months_half / 2

# Printing the result
print(f"Average number of '0' appearances per 12 months for users with at least one '0': {average_zeros_per_12_months}")
```

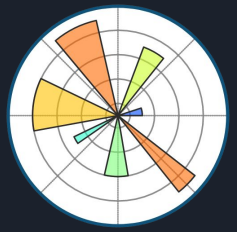
```
Average number of '0' appearances per 12 months for users with at least one '0': 6.471897774771895
```

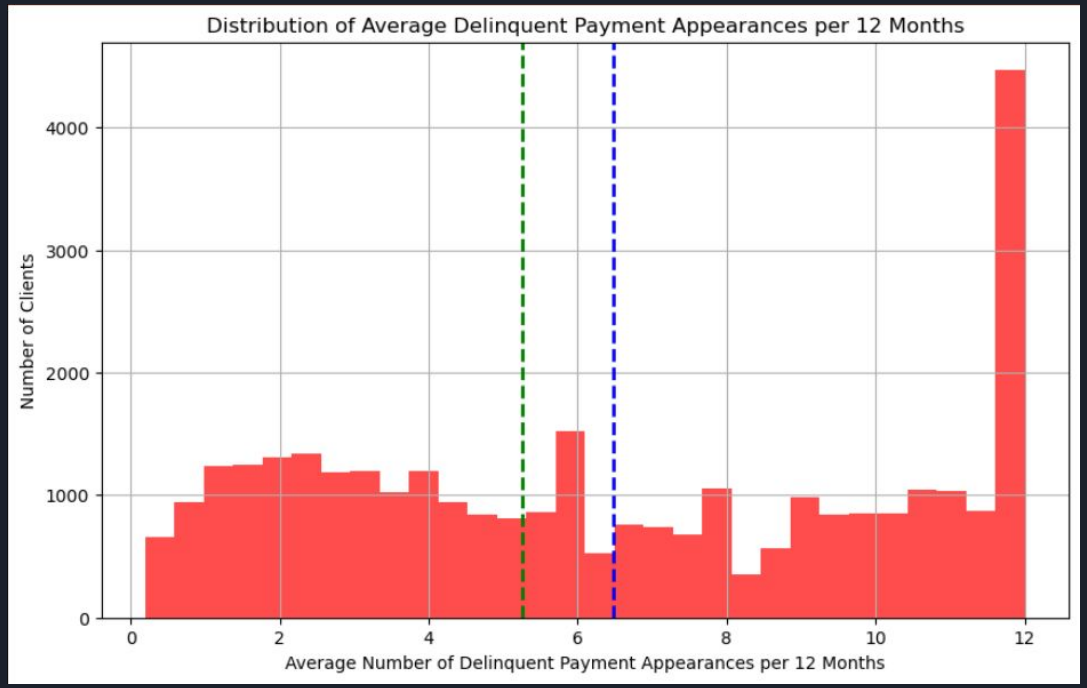
Given this average, we realize these people are, on average, very delinquent. Further filtering is needed. This number alone was not enough for me to make a decision with. I did use it up until i began tuning the model. Then came back to refactor…

Data Filtering and Exploratory Data Analysis

*Building on the previous slide

Revisiting the 0 user data, I generated this visualization

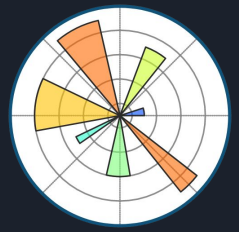


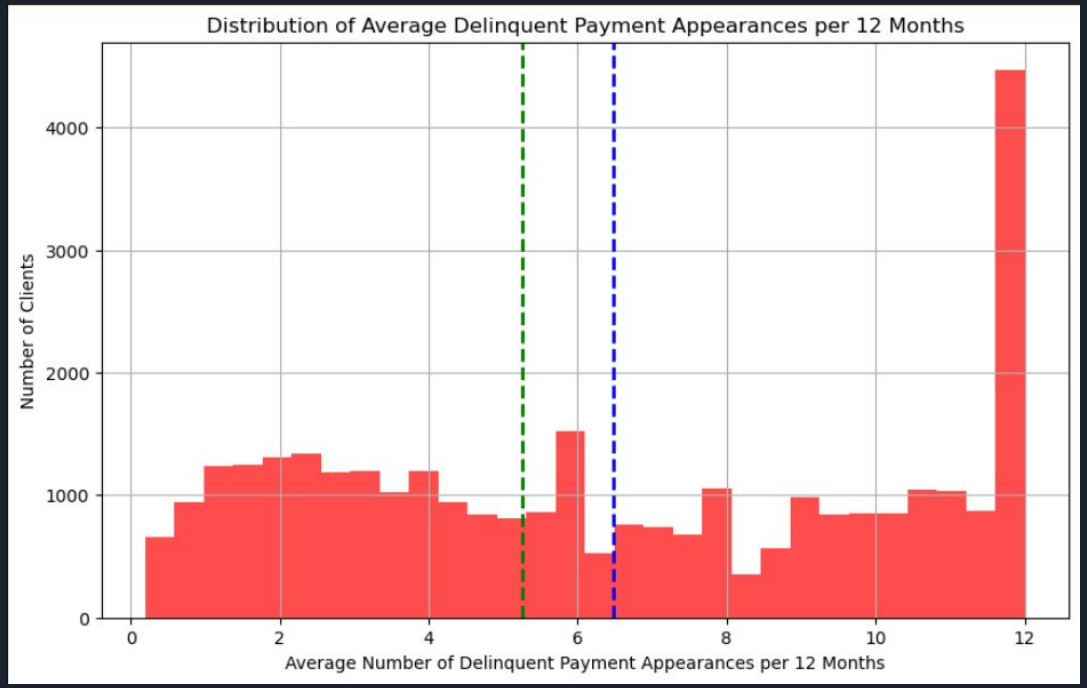


Data Filtering and Exploratory Data Analysis

*Building on the previous slide

The blue line represents the average. Notice the large spike around the 6th occurrence?





Data Filtering and Exploratory Data Analysis

*Wrapping it up

After writing some code that applied these filters to the dataframe, and creating the [‘Target’] y column, the number of targets reduced from 766,140 to 251,430.

Originally, my first shot through I only allowed C and X users into the target class. This resulted in around 18,668 targets and more than 750,000 bogeys.

This led me to a model with 98% accuracy. Too good to be true. The model would decline all new applicants.

This showed me the next challenge, handling the imbalanced data.

```
frequncy = df['Target'].value_counts()
frequncy
```

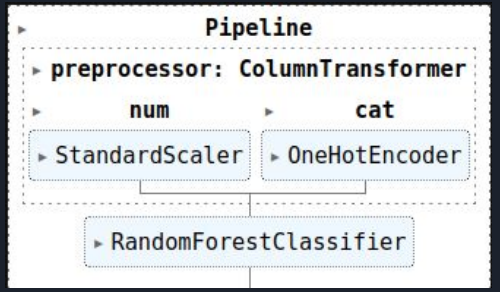
```
Target
0.0    526285
1.0    251430
Name: count, dtype: int64
```

Since the targets were only 2.4% of the training data, the model could achieve a 97.6% accuracy by just declining everyone.

This led me to reiterate, and allow for some of the 0 class into the target class, since they are a significant portion of the data set.

Training, and Handling the Imbalanced Data

I decided to build a pipeline to handle the preprocessing and the Model.



Handling the imbalance data was simple.

I considered using a method like SMOTE to oversample the minority class.

However there was a simple solution, a parameter in the RFC model.

class_weight

```
# Step 3: Define the model
model = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier(class_weight='balanced', random_state=42))
])
```

Tuning the Model

A few approaches take to tune:

Adjusting the decision threshold

Adjusting the Delinquent payment threshold

Adjusting the months on book threshold

Allowing for those 0 class users in the first place

Balancing the imbalance training data

Tuning the Model

Adjusting the decision threshold

This is the first evaluation of the model.

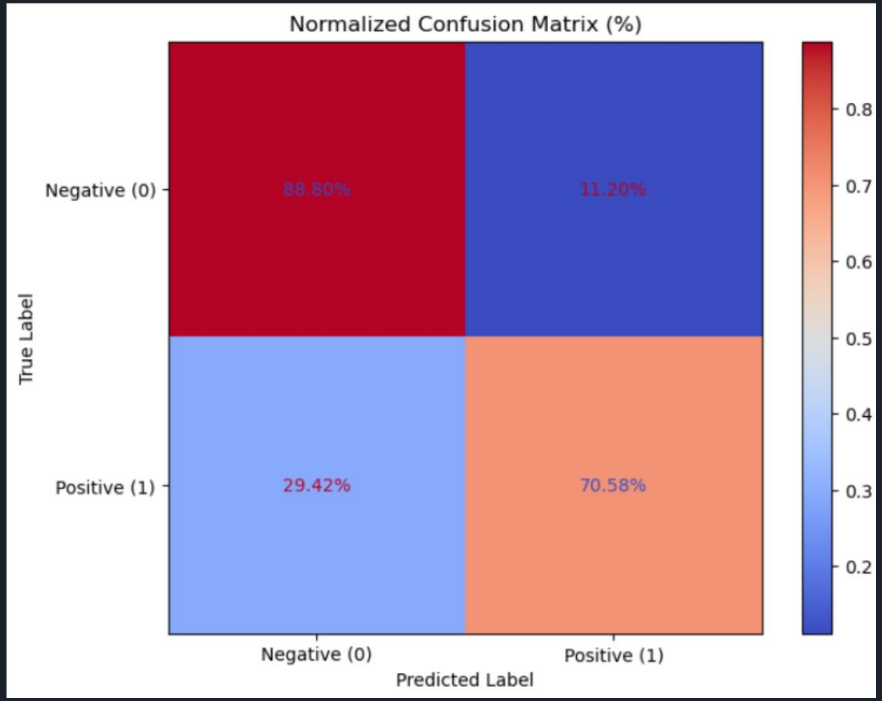
Good true negatives

Acceptable False positives

Poor true positives

Horrible false negatives

I believe that false negatives (recall) is very important in this context, we do not want to turn away our targets.



Tuning the Model

Adjusting the decision threshold

In an attempt to improve recall, I begin to adjust the decision threshold.

Lowering it to allow for more positive predictions.

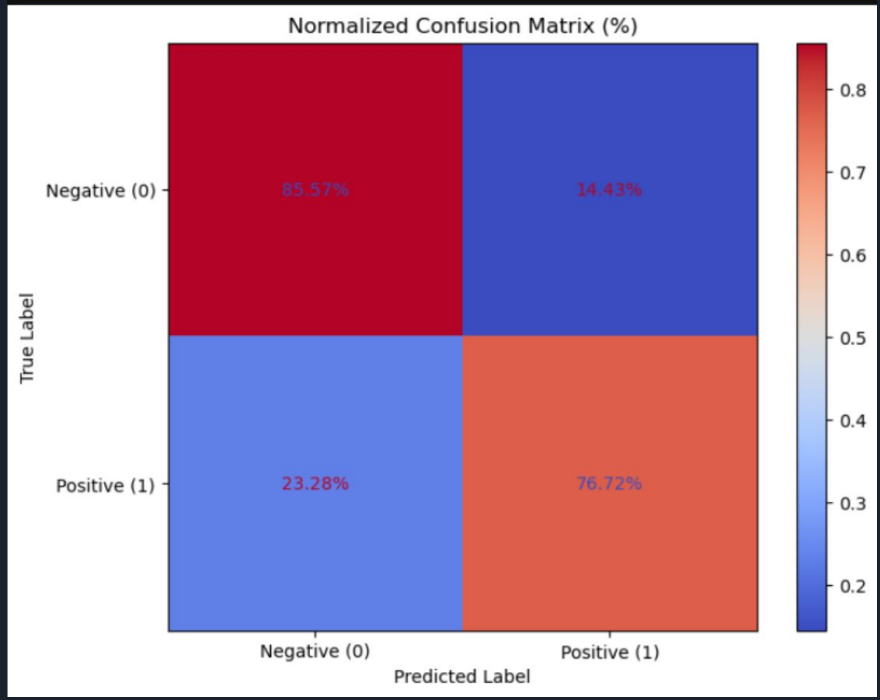
Lowering it from a 0.5 to a 0.45 improved recall, as well as true negatives and true positives.

False positives are safe, and less risky, as these clients can be terminated after displaying poor behavior.

But turning away targets, will hurt us much more, as we lose their long term gains, over the FP short term losses. This is a good trade.

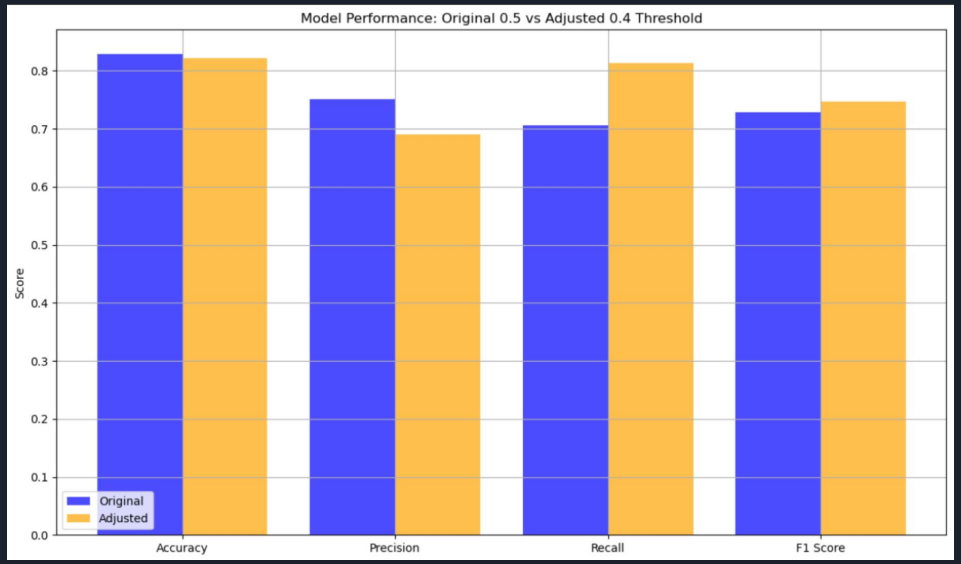
I am still not happy with the model at this point though.

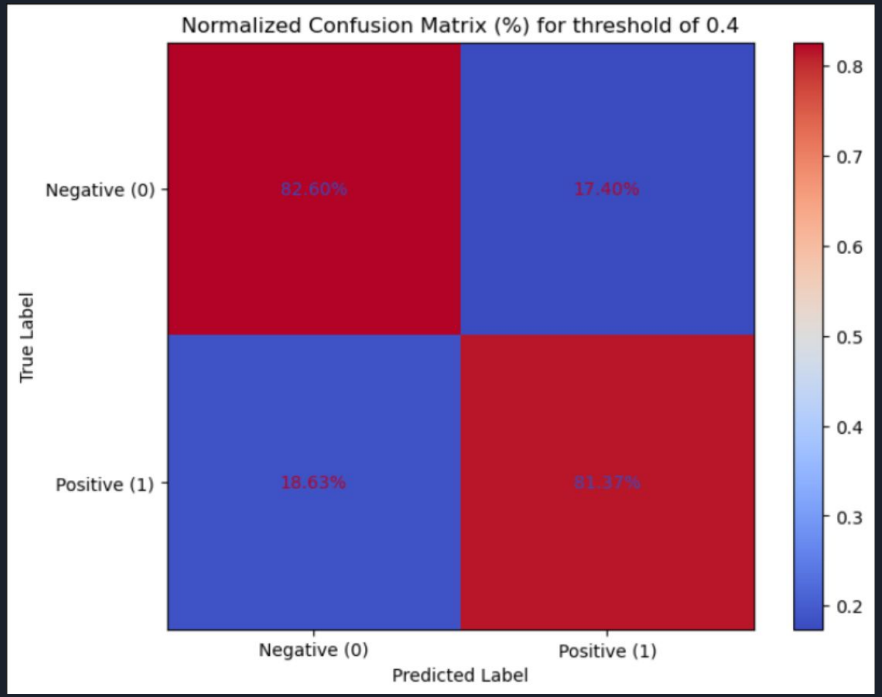
```
# tuned threshold to 0.45
```



Tuning the Model

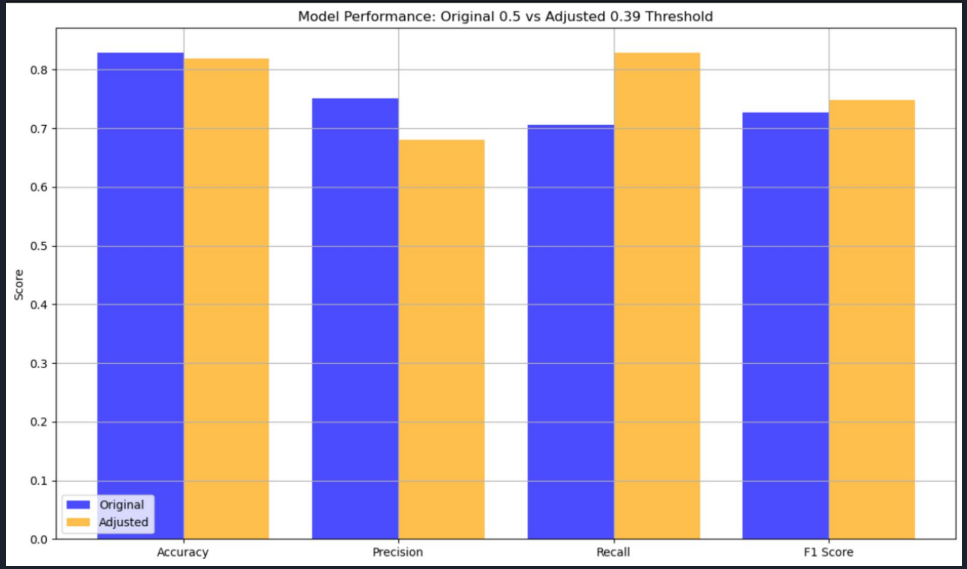
Adjusting the decision threshold

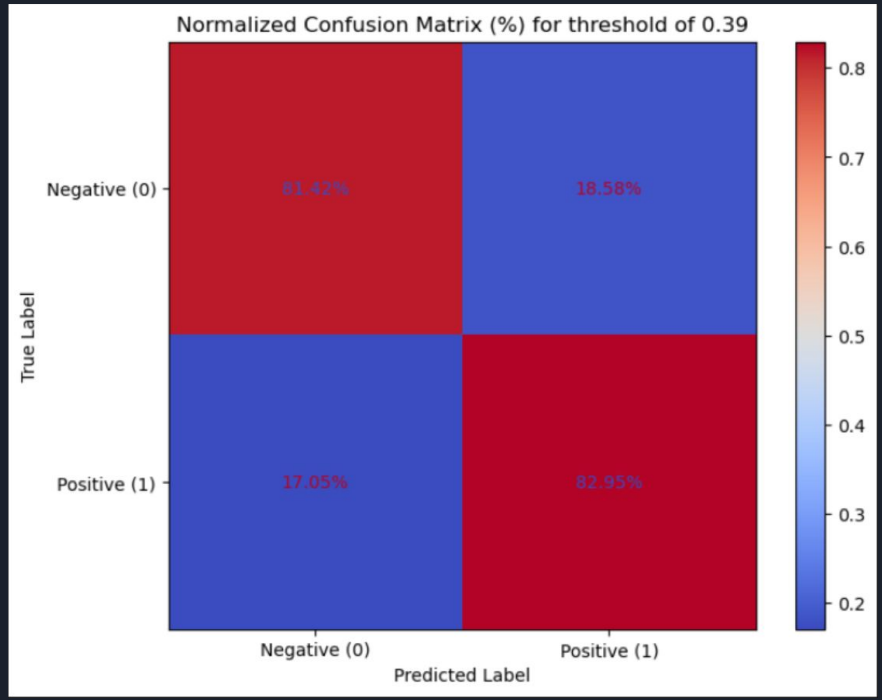




Tuning the Model

Adjusting the decision threshold





Tuning the Model

Adjusting the decision threshold, balancing the data

These evaluations were calculated on imbalance data.

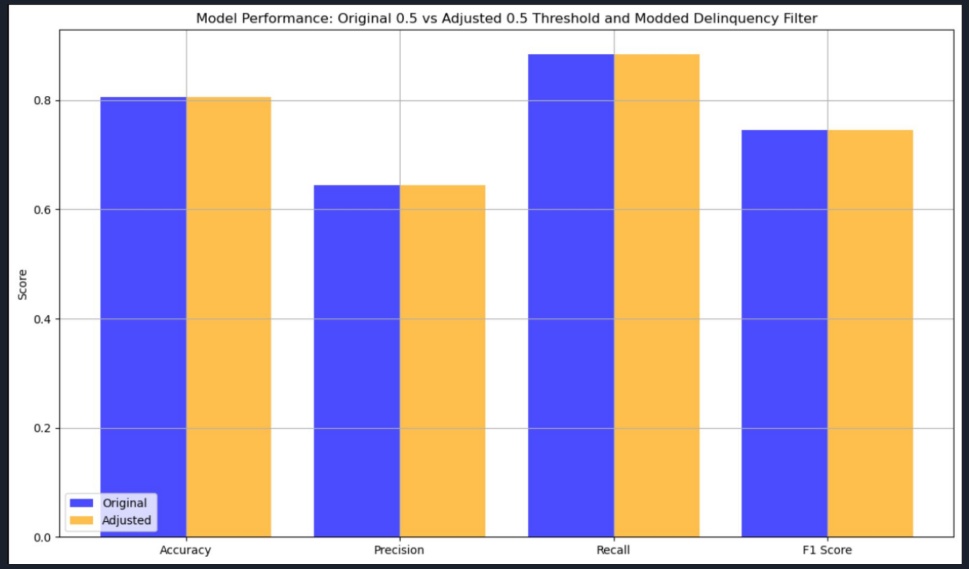
As well as the Average Delinquent Payment Appearance average at the average, before adjusting to the preferred visual value.

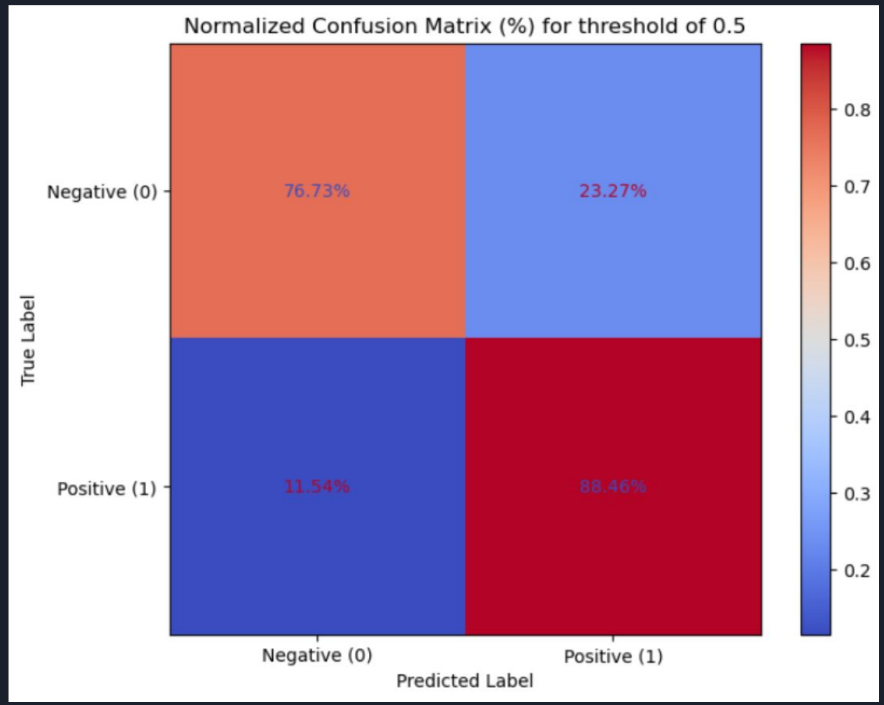
Next are the results after applying class_weights, and adjusting that average.

```
# Step 3: Define the model
model = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier(class_weight='balanced', random_state=42))
])
```

Tuning the Model

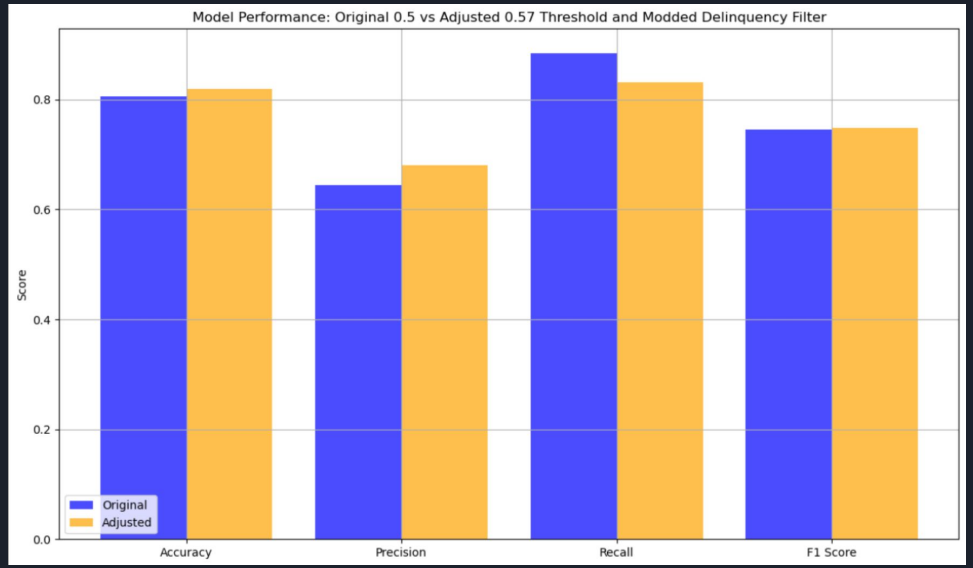
Adjusting the decision threshold, balanced data, delinquency filter

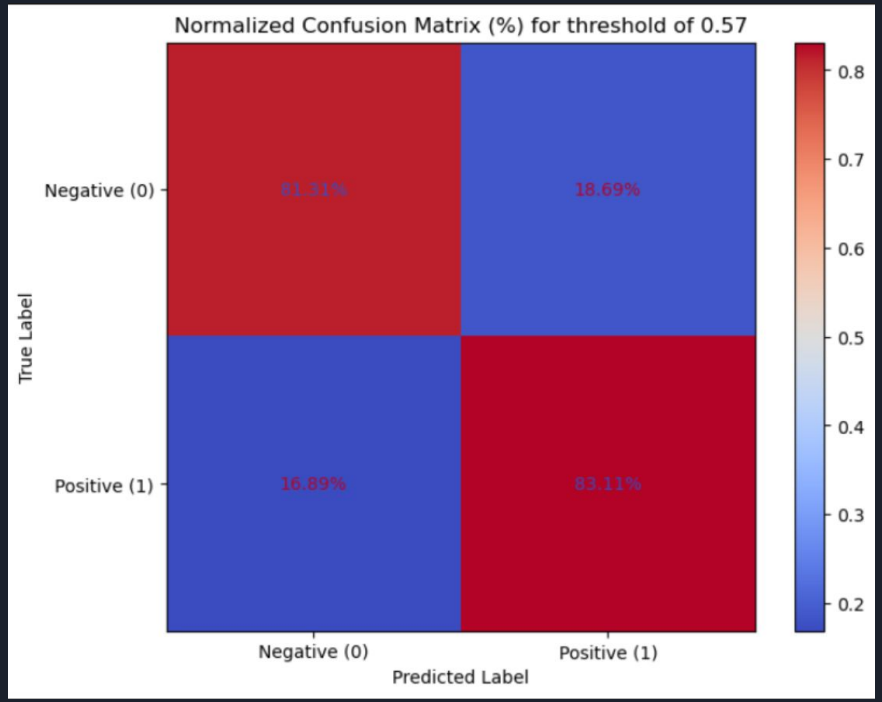




Tuning the Model

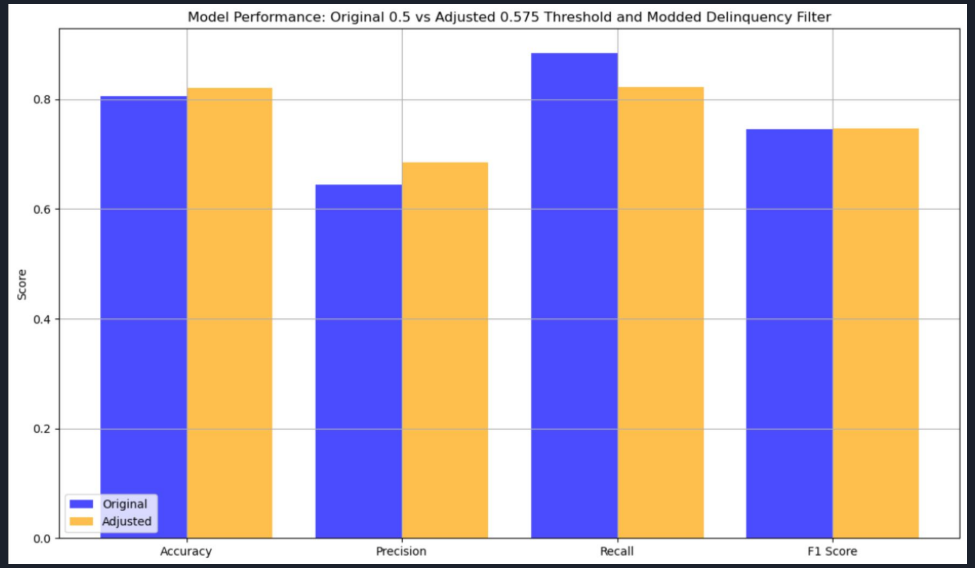
Adjusting the decision threshold, balanced data

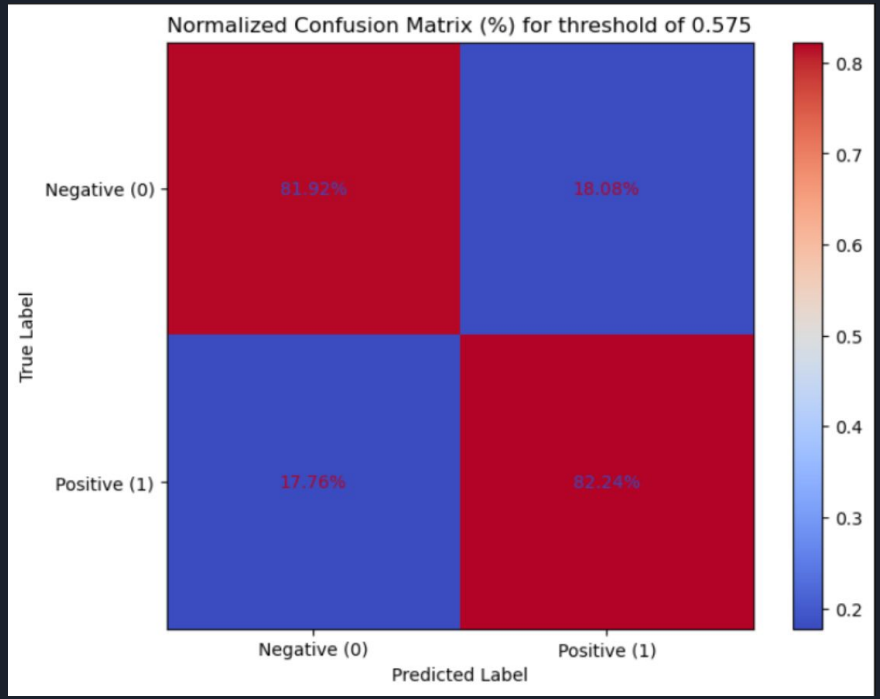




Tuning the Model

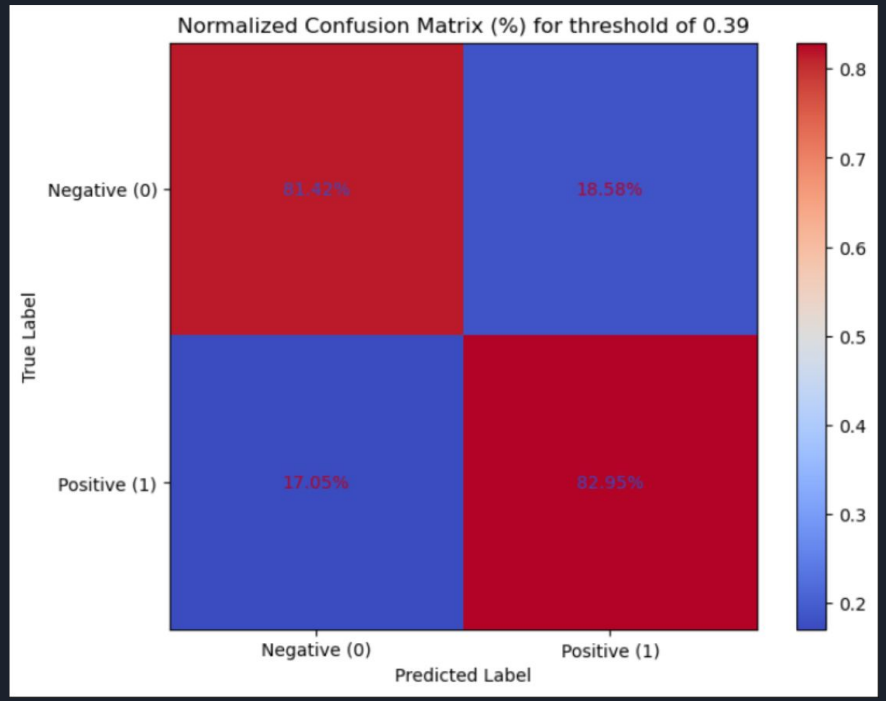
Adjusting the decision threshold, balanced data

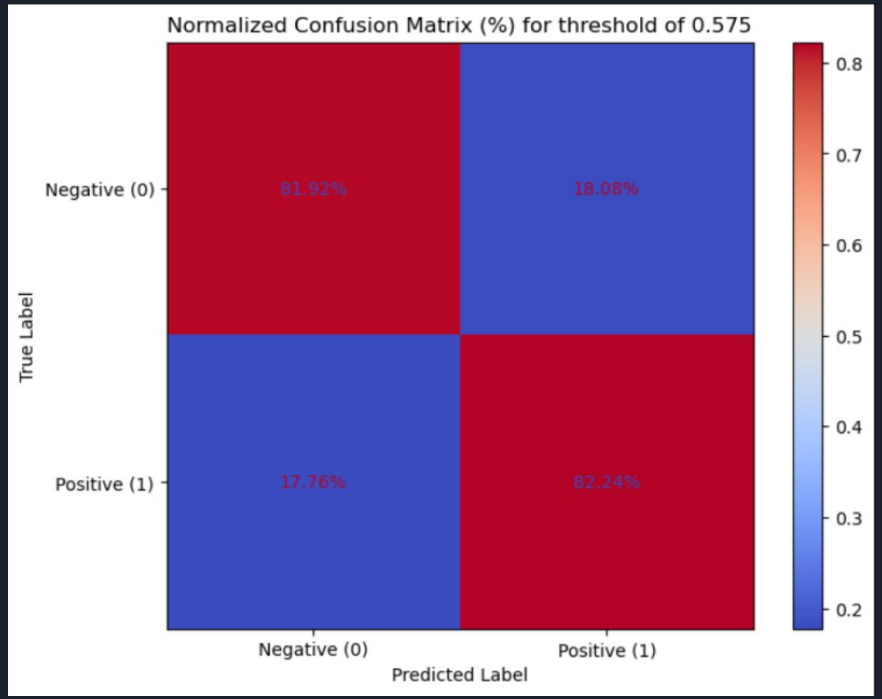




Tuning the Model

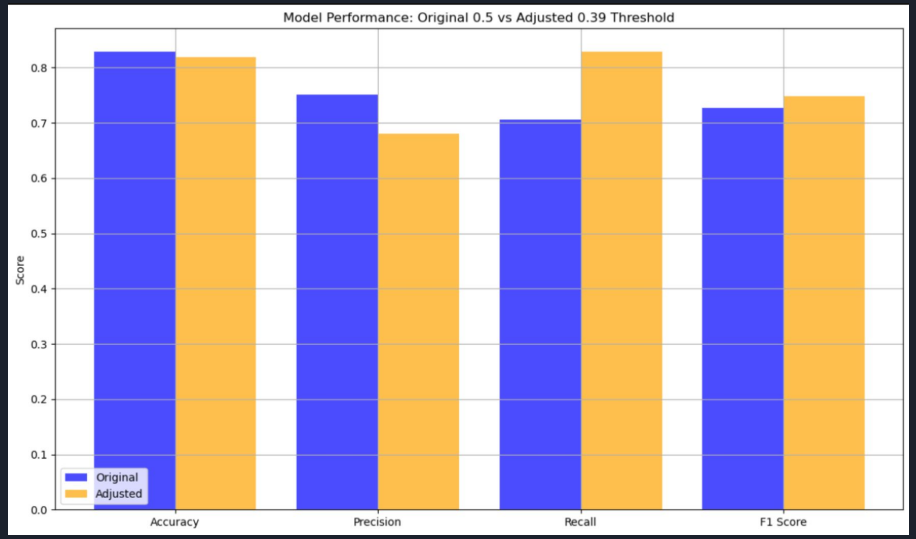
Comparing results to results

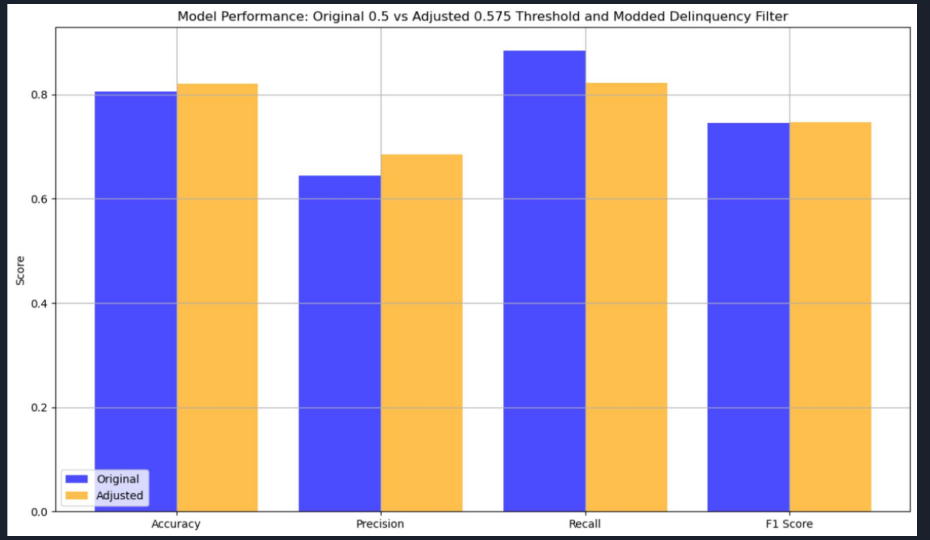




Tuning the Model

Comparing results to results





Tuning the Model

Wrapping it up

The only interesting distinction between before and after was the difference in threshold.

To get similar performance, we needed to raise the threshold from 39 to 57.5.

Given more time, I would like to experiment with oversampling the data using SMOTE, instead of class_weight

I would also like to try a hybrid sampling approach.

I would wrap most of the code in a function that will iterate through all combinations of parameters.

I would also like to try applying a margin around the threshold, which would render a range as unclassifiable. EG: sending those applicants to a human for a review. This i believe would reduce both the False Positive Rate and the False Negative Rate. Only allowing the model to make more certain decisions.

Live Demonstration

I have stored the model on disk, and built a UI to simulate the deployment of this model in the real world.

Questions?