Note: We removed all the irrelevant cells, so that you can easily see our model results

Data balancing

Upsampling

```
# Upsample the minority class ('Charged Off') in the training set
charged_off_train = X_train[y_train == 0]  # Charged Off
fully_paid_train = X_train[y_train == 1]  # Fully Paid

charged_off_upsampled = resample(charged_off_train,
                                 replace=True,
                                 n_samples=len(fully_paid_train),
                                 random_state=42)

# Combine upsampled minority with majority in the training set
X_train_upsampled = pd.concat([fully_paid_train,
charged_off_upsampled])
y_train_upsampled = pd.concat([y_train[y_train == 1], pd.Series([0] *
len(charged_off_upsampled))])


print("Before Upsampling:")
print(y_train.value_counts())


print("\nAfter Upsampling:")
print(y_train_upsampled.value_counts())
```

```
Before Upsampling:
1    50909
0    10127
Name: Loan Status, dtype: int64

After Upsampling:
1    50909
0    50909
dtype: int64
```

```
X_train_upsampled
```

```
       Credit Score  Annual Income  Current Loan Amount  \
34602         691.0       817988.0             161194.0
26381         692.0      1488840.0             333674.0
31618         669.0       714286.0             272932.0
5335          720.0      1456065.0             324236.0
```

```
26277          743.0       929480.0              302962.0
...              ...            ...                   ...
6696           675.0       940120.0              172546.0
65671          736.0       301663.0               87318.0
20985          719.0       583908.0              214104.0
20962          709.0      1802530.0              174702.0
19867          746.0      1532160.0              532224.0

       Years of Credit History       LTI        Term Home Ownership
34602                     21.5  0.197062   Long Term  Home Mortgage
26381                     14.1  0.224117   Long Term           Rent
31618                     24.5  0.382105   Long Term       Own Home
5335                      33.1  0.222680   Long Term  Home Mortgage
26277                     17.4  0.325948  Short Term           Rent
...                        ...       ...         ...            ...
6696                       9.8  0.183536   Long Term       Own Home
65671                     49.0  0.289455  Short Term       Own Home
20985                      7.3  0.366674  Short Term           Rent
20962                     18.5  0.096920  Short Term       Own Home
19867                     12.6  0.347368  Short Term  Home Mortgage

[101818 rows x 7 columns]
```

Machine Learning

Models chosen and it's considerations

Dummy classfier – serves as a simple baseline to compare against other more complex classifiers

Logistic Regression – Simple, interpretable, but may struggle with non-linear relationships

Random Forest – An ensemble method, reduces overfitting, and handles imbalanced data well

XG boost – is a powerful gradient boosting algorithm renowned for its accuracy and ability to handle complex relationships in data

Dummy Classifier

We chose dummy classifier to act as our baseline model.

```
dummy = DummyClassifier(strategy='uniform')
dummy.fit(X_train_upsampled, y_train_upsampled)

# Predict on test data (original distribution)
y_pred_dummy = dummy.predict(X_test)
```

```
# Evaluate
print(classification_report(y_test, y_pred_dummy))
```

```
              precision    recall  f1-score   support

           0       0.16      0.49      0.24      2532
           1       0.83      0.50      0.63     12727

    accuracy                           0.50     15259
   macro avg       0.50      0.49      0.43     15259
weighted avg       0.72      0.50      0.56     15259
```

Logistic Regression

We chose logistic regresion instead of linear as we want to predict categorical outcomes (loan default or repayment). Whereas linear Regression is more suitable for predicting continuous or numerical values

```
numeric_cols
```

```
['Credit Score',
 'Annual Income',
 'Current Loan Amount',
 'Years of Credit History',
 'LTI']
```

```
# One-hot encode categorical variables
X_train_up_encoded = pd.get_dummies(X_train_upsampled,
drop_first=True)
X_test_encoded = pd.get_dummies(X_test, drop_first=True)

# Align columns
X_train_up_encoded, X_test_encoded =
X_train_up_encoded.align(X_test_encoded, join='outer', axis=1,
fill_value=0)

param_grid = {
    'C': [0.01, 0.1, 1, 10, 100],                 # Regularization
strength
    'penalty': ['l1', 'l2'],                      # L1 or L2
regularization
    'solver': ['liblinear'],                      # 'liblinear' supports
both l1 and l2
}
```

```
grid_search = GridSearchCV(LogisticRegression(max_iter=500),
                           param_grid=param_grid,
                           scoring='f1_macro',
                           cv=5,
                           n_jobs=-1,
                           verbose=1)

grid_search.fit(X_train_up_encoded, y_train_upsampled)

y_pred_best_lr = grid_search.best_estimator_.predict(X_test_encoded)


print("Best parameters:", grid_search.best_params_)
print("Best Logistic Regression Performance (Upsampled + Encoded +
Standardized):")
print(classification_report(y_test, y_pred_best_lr))
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters: {'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}
Best Logistic Regression Performance (Upsampled + Encoded +
Standardized):
              precision    recall  f1-score   support

           0       0.20      0.68      0.31      2532
           1       0.88      0.45      0.59     12727

    accuracy                           0.49     15259
   macro avg       0.54      0.57      0.45     15259
weighted avg       0.76      0.49      0.55     15259
```

We used "macro_f1" as our scoring metric as we want to give equal weight to each class (simple average of F1s). Because we have a class imbalance.

Random Forest

```
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2],
    'max_features': ['sqrt', 'log2']
}


rf = RandomForestClassifier(random_state=42)


grid_search = GridSearchCV(
```

```
    estimator=rf,
    param_grid=param_grid,
    cv=3,
    n_jobs=-1,
    scoring='f1_macro',
    verbose=2
)


grid_search.fit(X_train_up_encoded, y_train_upsampled)

# Get the best model
best_rf = grid_search.best_estimator_

# Predict on the aligned test set
y_pred_best_rf = best_rf.predict(X_test_encoded)

# Output results
print("Best Hyperparameters (Upsampled RF):",
grid_search.best_params_)
print("Random Forest Performance (Upsampled Data):")
print(classification_report(y_test, y_pred_best_rf))
```

```
Fitting 3 folds for each of 48 candidates, totalling 144 fits
Best Hyperparameters (Upsampled RF): {'max_depth': None,
'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2,
'n_estimators': 200}
Random Forest Performance (Upsampled Data):
              precision    recall  f1-score   support

           0       0.33      0.12      0.17      2532
           1       0.84      0.95      0.89     12727

    accuracy                           0.81     15259
   macro avg       0.59      0.53      0.53     15259
weighted avg       0.76      0.81      0.78     15259
```

XG BOOST

```
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [3, 6],
    'learning_rate': [0.05, 0.1, 0.2],
    'subsample': [0.8, 1.0],
    'colsample_bytree': [0.8, 1.0],
    'gamma': [0,1],
    'min_child_weight': [1,3]
}
```

```python
# Initialize XGBoost classifier
xgb_clf = XGBClassifier(
    random_state=42,
    use_label_encoder=False,
    eval_metric='logloss'
)

# Grid search setup
grid_search = GridSearchCV(
    estimator=xgb_clf,
    param_grid=param_grid,
    scoring='f1_macro',
    cv=2,  # Faster cross-validation
    n_jobs=-1,
    verbose=2
)

# Fit model on upsampled + encoded training data
grid_search.fit(X_train_up_encoded, y_train_upsampled)

# Predict on aligned and encoded test set
best_xgb = grid_search.best_estimator_
y_pred_best_xgb = best_xgb.predict(X_test_encoded)

# Output results
print("Best Hyperparameters (Upsampled XGBoost):",
grid_search.best_params_)
print("XGBoost Performance (Upsampled Data):")
print(classification_report(y_test, y_pred_best_xgb,
target_names=["Charged Off", "Fully Paid"]))
```

```
Fitting 2 folds for each of 192 candidates, totalling 384 fits
Best Hyperparameters (Upsampled XGBoost): {'colsample_bytree': 1.0,
'gamma': 1, 'learning_rate': 0.2, 'max_depth': 6, 'min_child_weight':
1, 'n_estimators': 200, 'subsample': 0.8}
XGBoost Performance (Upsampled Data):
              precision    recall  f1-score   support

 Charged Off       0.23      0.65      0.34      2532
  Fully Paid       0.89      0.57      0.70     12727

    accuracy                           0.58     15259
   macro avg       0.56      0.61      0.52     15259
weighted avg       0.78      0.58      0.64     15259
```