# CREDIT CARD APPROVAL PREDICTION

Machine Learning

ABSTRACT

Machine Learning Models to predict whether an applicant is a good applicant or has bad credit

FAZELEH, MALIHA, JYOTHI AND STANLEY

# Contents

# 1. Project Summary

This is a Machine Learning project which is meant to predict whether an applicant is a good or a bad client. Models will be built using the personal information of customers and their historical tendencies to pay back loans. Predictions will be made using labels that are not found in any of the datasets.

There is flexibility there as we are able to use any formula to generate corresponding binary credit status. Credit scores of either 0 or 1 will be generated and used as target labels.

# 2. File system

    .gitignore
    decision_tree.ipynb
    logistic_regression.ipynb
    neural_network.ipynb
    pre_processing.ipynb
    random_forest.ipynb
    README.md

      +---Output
          credit_decision_tree_optimized.pdf
          credit_decision_tree_optimized.png
          credit_decision_tree_preoptimized.pdf
          credit_decision_tree_preoptimized.png
          full_data.csv
          pre_encoded.csv

      +---Resources
          application_record.csv
          credit_record.csv

# 3. Preprocessing

## 3.1 Datasource

Two .csv datasets (application_record.csv and credit_record.csv) were used for the project. They retrieved from Kaggle via the url:

> https://www.kaggle.com/datasets/rikdifos/credit-card-approval-prediction/download?datasetVersionNumber=3

### 3.1.1 Application Record

Contains application records of customers and their personal information to be considered as the features to enable prediction. There is an ID column that enables merging with credit_record.csv.

### 3.1.2 Credit record

It contains previous credit records of customers in "application_record.csv". It contains previous monthly loan repayment records for customers. There is an ID column that enables merging with application_record.csv.

The two datasets will be subsequently merged enabling us to extract features and target labels for model building.

## 3.2   Limitations of Datasets

- One major challenge of these two data sets is that there is no distinctive target column for either "Good Credit" or "Bad Credit". To solve this issue, Binary values will be deduced using the credit record dataset.
- Also, the data is heavily unbalanced potentially adding to the need to apply optimization if the desired accuracy is not achieved.

## 3.3   Data Base Engineering

Data is extracted and loaded in their raw form into the database for scalability purposes which will allow for prospective applications and credit records to be added in the future.

Tools:

- PostgreSQL

## Loading Data Frames into DB as Tables

```python
protocol = 'postgresql'
username = 'postgres'
password = pw
host = 'localhost'
port = 5432
database_name = 'creditCheck_db'
rds_connection_string = f'{protocol}://{username}:{password}@{host}:{port}/{database_name}'
engine = create_engine(rds_connection_string)

Base = declarative_base()
```

```python
# Creating poke table
class credit_record(Base):
    extend_existing=True
    __tablename__ = "credit_record"

    cr_id = Column("CR_ID", Integer, primary_key = True)
    id_ = Column("ID", Integer)
    month_balance = Column("MONTHS_BALANCE", Integer)
    status = Column("STATUS", String)
```

## 3.4   Credit Record Engineering
The relevant columns in the credit record Table are the "ID", "MONTHS_BALANCE", and "STATUS" columns.

### Credit Record Engineering

```python
credit_record_df.head()
```

|   | CR_ID | ID | MONTHS_BALANCE | STATUS |
|---|-------|----|--------------|--------|
| 0 | 0 | 5001711 | 0 | X |
| 1 | 1 | 5001711 | -1 | 0 |
| 2 | 2 | 5001711 | -2 | 0 |
| 3 | 3 | 5001711 | -3 | 0 |
| 4 | 4 | 5001712 | 0 | C |

The dataset has a "STATUS" column which contains values with these explanations.

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-offs for more than 150 days
C: paid off that month
X: No loan for the month

### 3.4.1    Converting STATUS values To Target Labels

We dropped all STATUS columns with "X" values as we couldn't conclude whether a customer with no loan for a particular month should be considered as a customer with good credit or bad credit for that month.

```python
print(credit_record_df.STATUS.unique())
```

```
['X' '0' 'C' '1' '2' '3' '4' '5']
```

```python
# Since the "X" value signifies "No loan for the month", this can be counted as irrelevant
# to whether or not they defaulted and if used could create bias
# There are a lot of account numbers with a month_balances with corresponding status labeled as "X" so can't be used
credit_record_df.drop(credit_record_df[credit_record_df["STATUS"] == "X"].index, inplace = True)
credit_record_df.head()
```

It became also clear that our target label values could be deduced using the following formula.

Good Credit = 0

Bad Credit = 1

```python
# Replace categorical credit status values with binary values
credit_record_df = credit_record_df.apply(lambda x: x.replace({'C': 0, '0' : 0, '1' : 1, '2' : 1, '3' : 1, '4' : 1, '5' : 1}, reg
display(credit_record_df['STATUS'].head())
display(credit_record_df['STATUS'].value_counts())
```

C: paid off that month = Good Credit = 0

0: 1-29 days past due = Good Credit = 0

All other STATUS values will be replaced with Bad Credit = 1.

The averages of all the monthly statuses of a customer were calculated and any status value of more than 0 was classed as 1 (Bad credit), whiles all 0 values were classed as 0 (Good credit). This way all float values will also be replaced with binary values.

```
# Replace float values with binaries
credit_record_df["Status2"] = credit_record_df["STATUS"].apply(lambda x: 1 if x > 0 else 0)
credit_record_df.rename(columns = {"STATUS" : "MONTH_BAL_AVG", "Status2" : "STATUS"}, inplace = True)
credit_record_df = credit_record_df.drop(columns="MONTH_BAL_AVG", axis=1)

credit_record_df.head()
```

|   | ID | STATUS |
|---|------|--------|
| 0 | 5001711 | 0 |
| 1 | 5001712 | 0 |
| 2 | 5001717 | 0 |
| 3 | 5001718 | 1 |
| 4 | 5001719 | 0 |

## 3.5   Application record data Engineering

The relevant columns in the application record Table are:
'CODE_GENDER', 'FLAG_OWN_CAR', 'FLAG_OWN_REALTY', 'CNT_CHILDREN', 'AMT_INCOME_TOTAL',
'NAME_INCOME_TYPE', 'NAME_EDUCATION_TYPE', 'NAME_FAMILY_STATUS', 'NAME_HOUSING_TYPE',
'DAYS_BIRTH', 'DAYS_EMPLOYED', 'FLAG_MOBIL', 'FLAG_WORK_PHONE', 'FLAG_PHONE', 'FLAG_EMAIL',
'OCCUPATION_TYPE' and 'CNT_FAM_MEMBERS' columns.

```
application_record_df.head()
```

|   | AR_ID | ID | CODE_GENDER | FLAG_OWN_CAR | FLAG_OWN_REALTY | CNT_CHILDREN | AMT_INCOME_TOTAL | NAME_INCOME_TYPE | NAME_EDUCATIO |
|---|-------|------|-------------|--------------|-----------------|--------------|------------------|------------------|----------------|
| 0 | 0 | 5008804 | M | Y | Y | 0 | 427500.0 | Working | Higher e |
| 1 | 1 | 5008805 | M | Y | Y | 0 | 427500.0 | Working | Higher e |
| 2 | 2 | 5008806 | M | Y | Y | 0 | 112500.0 | Working | Secondary / se |
| 3 | 3 | 5008808 | F | N | Y | 0 | 270000.0 | Commercial associate | Secondary / se |
| 4 | 4 | 5008809 | F | N | Y | 0 | 270000.0 | Commercial associate | Secondary / se |

### 3.5.1   Merging both datasets and dropping irrelevant columns

Both data sets were merged and the irrelevant columns were dropped finally.

## Merging both datasets

```
# Merge the two datasets keeping only IDs common to both dataframes
full_df = application_record_df.merge(credit_record_df, on=['ID'], how='inner')
display(full_df.head().T)
full_df.shape
```

|   | 0 | 1 | 2 | 3 | 4 |
|---|------|------|------|------|------|
| AR_ID | 0 | 1 | 2 | 3 | 5 |
| ID | 5008804 | 5008805 | 5008806 | 5008808 | 5008810 |
| CODE_GENDER | M | M | M | F | F |
| FLAG_OWN_CAR | Y | Y | Y | N | N |
| FLAG_OWN_REALTY | Y | Y | Y | Y | Y |
| CNT_CHILDREN | 0 | 0 | 0 | 0 | 0 |
| AMT_INCOME_TOTAL | 427500.0 | 427500.0 | 112500.0 | 270000.0 | 270000.0 |
| NAME_INCOME_TYPE | Working | Working | Working | Commercial associate | Commercial associate |
| NAME_EDUCATION_TYPE | Higher education | Higher education | Secondary / secondary special | Secondary / secondary special | Secondary / secondary special |
| NAME_FAMILY_STATUS | Civil marriage | Civil marriage | Married | Single / not married | Single / not married |
| NAME_HOUSING_TYPE | Rented apartment | Rented apartment | House / apartment | House / apartment | House / apartment |
| DAYS_BIRTH | -12005 | -12005 | -21474 | -19110 | -19110 |

### 3.5.2　Encoding Categorical columns

```python
# Encoding NAME_INCOME_TYPE column
name_income_type_mapper = {'Working': 2, 'Commercial associate': 2, 'Pensioner': 1, 'State servant': 3, 'Student': 0}
pre_encoded_df["NAME_INCOME_TYPE"] = pre_encoded_df["NAME_INCOME_TYPE"].replace(name_income_type_mapper)

#  Encoding NAME_EDUCATION_TYPE column
name_education_type_mapper = {'Higher education': 3, 'Secondary / secondary special': 1, 'Incomplete higher': 2,
 'Lower secondary': 0, 'Academic degree': 3}
pre_encoded_df["NAME_EDUCATION_TYPE"] = pre_encoded_df["NAME_EDUCATION_TYPE"].replace(name_education_type_mapper)

#  Encoding NAME_FAMILY_STATUS column
name_family_status_mapper = {'Civil marriage': 3, 'Married': 4, 'Single / not married': 0, 'Separated': 1, 'Widow': 2}
pre_encoded_df["NAME_FAMILY_STATUS"] = pre_encoded_df["NAME_FAMILY_STATUS"].replace(name_family_status_mapper)

#  Encoding NAME_HOUSING_TYPE column
name_housing_type_mapper = {'Rented apartment': 1, 'House / apartment': 2, 'Municipal apartment': 1, 'With parents': 0,
```

```python
# Encoding binary categorical (CODE_GENDER, FLAG_OWN_CAR, FLAG_OWN_REALTY) variables using get_dummies
final_df = pd.get_dummies(pre_encoded_df)

# Move the status index to the end
final_df = final_df.reindex(columns = [col for col in final_df.columns if col != 'STATUS'] + ['STATUS'])
final_df.head().T
```

|  | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| **CNT_CHILDREN** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **AMT_INCOME_TOTAL** | 427500.0 | 427500.0 | 112500.0 | 270000.0 | 270000.0 |
| **NAME_INCOME_TYPE** | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 |
| **NAME_EDUCATION_TYPE** | 3.0 | 3.0 | 1.0 | 1.0 | 1.0 |
| **NAME_FAMILY_STATUS** | 3.0 | 3.0 | 4.0 | 0.0 | 0.0 |
| **NAME_HOUSING_TYPE** | 1.0 | 1.0 | 2.0 | 2.0 | 2.0 |
| **DAYS_BIRTH** | -12005.0 | -12005.0 | -21474.0 | -19110.0 | -19110.0 |
| **DAYS_EMPLOYED** | -4542.0 | -4542.0 | -1134.0 | -3051.0 | -3051.0 |

### 3.5.3　Exporting final dataframe to a .csv file for use in model engineering

```python
print(f"Good Credit: {len(final_df[final_df['STATUS'] == 0])}")
print(f"Bad Credit: {len(final_df[final_df['STATUS'] == 1])}")
```

```
Good Credit: 28819
Bad Credit: 4291
```

```python
final_df.to_csv("Output/full_data.csv", index=False)
```

# 4. Model Engineering

## 4.1 Supervised Learning

Since we now have target labels, we will be building supervised learning labels.

### 4.1.1 Classification Method

### 4.1.2 Binary classification

There are two class labels (Good Credit "0" and Bad Credit "1") for our classification.

### 4.1.3 Model selection

We are considering 4 Models for our project utilizing a mix of different optimization methods to enhance our efficiency.

1. Random Forest
2. Decision Tree
3. Logistic Regression
4. Neural Network

#### 4.1.3.1 Random Forest

#### 4.1.3.1.1 Training with the preoptimized or non-resampled dataset

```
# Displaying results
print("Confusion Matrix")
display(cm_df)
print(f"Accuracy Score : {acc_score}")
print("Classification Report")
print(classification_report(y_test, predictions))
```

Confusion Matrix

|          | Predicted 0 | Predicted 1 |
|----------|-------------|-------------|
| Actual 0 | 6958        | 286         |
| Actual 1 | 692         | 342         |

```
Accuracy Score : 0.8818555206571635
Classification Report
              precision    recall  f1-score   support

           0       0.91      0.96      0.93      7244
           1       0.54      0.33      0.41      1034

    accuracy                           0.88      8278
   macro avg       0.73      0.65      0.67      8278
weighted avg       0.86      0.88      0.87      8278
```
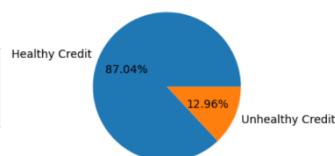
#### 4.1.3.1.2 Training with the optimized or non-resampled dataset

**Checking for imbalance**

```
# Checking for Balance
counter = Counter(y)
print(counter)
print(f"Heavily imbalanced: {round((np.sum(y) / len(y)) * 100, 2)}%")
```

```
Counter({0: 28819, 1: 4291})
Heavily imbalanced: 12.96%
```
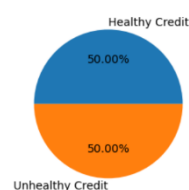


**Fixing data imbalance**

```
# transform the dataset
oversample = SMOTE()
X, y = oversample.fit_resample(X, y)
```

```
# summarize the new class distribution
counter_resampled = Counter(y)
print(counter_resampled)
```

```
Counter({1: 28819, 0: 28819})
```

After Resampling, the accuracy score is **0.92** which is an improvement over the **0.88** for preoptimization

```
# Displaying results
print("Confusion Matrix")
display(cm_df)
print(f"Accuracy Score : {acc_score}")
print("Classification Report")
print(classification_report(y_test, predictions))
```

Confusion Matrix

|          | Predicted 0 | Predicted 1 |
|----------|-------------|-------------|
| Actual 0 | 6648        | 573         |
| Actual 1 | 637         | 6552        |

```
Accuracy Score : 0.916030534351145
Classification Report
              precision    recall  f1-score   support

           0       0.91      0.92      0.92      7221
           1       0.92      0.91      0.92      7189

    accuracy                           0.92     14410
   macro avg       0.92      0.92      0.92     14410
weighted avg       0.92      0.92      0.92     14410
```

### 4.1.3.2    Decision Tree

#### 4.1.3.2.1    Training with the preoptimized or non-resampled dataset

```
# Displaying results
print("Confusion Matrix")
display(cm_df)
print(f"Accuracy Score : {acc_score}")
print("Classification Report")
print(classification_report(y_test, predictions))
```

Confusion Matrix

|          | Predicted 0 | Predicted 1 |
|----------|-------------|-------------|
| Actual 0 | 6865        | 379         |
| Actual 1 | 673         | 361         |

```
Accuracy Score : 0.8729161633244745
Classification Report
              precision    recall  f1-score   support

           0       0.91      0.95      0.93      7244
           1       0.49      0.35      0.41      1034

    accuracy                           0.87      8278
   macro avg       0.70      0.65      0.67      8278
weighted avg       0.86      0.87      0.86      8278
```
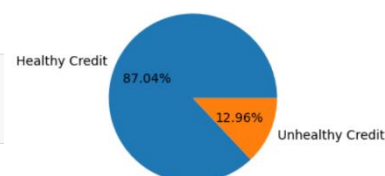
#### 4.1.3.2.2    Training with the optimized or non-resampled dataset

**Checking for imbalance**

```
# Checking for Balance
counter = Counter(y)
print(counter)
print(f"Heavily imbalanced: {round((np.sum(y) / len(y)) * 100, 2)}%")
```



```
Counter({0: 28819, 1: 4291})
Heavily imbalanced: 12.96%
```

**Fixing data imbalance**

```
# transform the dataset
oversample = SMOTE()
X, y = oversample.fit_resample(X, y)
```

```
# summarize the new class distribution
counter_resampled = Counter(y)
print(counter_resampled)
```



```
Counter({1: 28819, 0: 28819})
```

After Resampling, the accuracy score is **0.90** which is an improvement over the **0.87** for preoptimization

```
# Displaying results
print("Confusion Matrix")
display(cm_df)
print(f"Accuracy Score : {acc_score}")
print("Classification Report")
print(classification_report(y_test, predictions))
```

Confusion Matrix

|  | Predicted 0 | Predicted 1 |
|---|---|---|
| **Actual 0** | 6531 | 690 |
| **Actual 1** | 769 | 6420 |

```
Accuracy Score : 0.8987508674531576
Classification Report
              precision    recall  f1-score   support

           0       0.89      0.90      0.90      7221
           1       0.90      0.89      0.90      7189

    accuracy                           0.90     14410
   macro avg       0.90      0.90      0.90     14410
```

### 4.1.3.3    Logistic Regression

4.1.3.3.1    Training with no Optimization

Preoptimization parameters

```
# Print the classification report for the model
print("Classification Report")
print(classification_report(y_test, testing_predictions))
```

```
Classification Report
              precision    recall  f1-score   support

           0       0.87      1.00      0.93      7219
           1       0.00      0.00      0.00      1059

    accuracy                           0.87      8278
   macro avg       0.44      0.50      0.47      8278
weighted avg       0.76      0.87      0.81      8278
```

The data is highly unbalanced as all were predicted as good loans.

```
# Generate a confusion matrix for the model
testing_matrix = confusion_matrix(y_test, testing_predictions)
testing_matrix_df = pd.DataFrame(
    testing_matrix, index=["Actual 0", "Actual 1"], columns=["Predicted 0", "Predicted 1"]
)
print("Confusion Matrix")
display(testing_matrix_df)
# Calculating the accuracy score
accuracy_score1 = accuracy_score(y_test, testing_predictions)
print(f"Accuracy Score : {accuracy_score1}")
```

Confusion Matrix

|  | Predicted 0 | Predicted 1 |
|---|---|---|
| **Actual 0** | 7219 | 0 |
| **Actual 1** | 1059 | 0 |

```
Accuracy Score : 0.8720705484416525
```

### 4.1.3.3.2    Using Class Weights to reduce imbalance

```python
# Print the balanced_accuracy score of the model
train_balanced_accuracy=balanced_accuracy_score(y_train, training_predictions)
print(f"Train Accuracy Score : {train_balanced_accuracy}")
test_balanced_accuracy=balanced_accuracy_score(y_test, testing_predictions)
print(f"Test Accuracy Score : {test_balanced_accuracy}")
```

```
Train Accuracy Score : 0.540184039237257
Test Accuracy Score : 0.5310482606687499
```

```python
# Calculating the accuracy score
accuracy_score1 = accuracy_score(y_test, testing_predictions)
print(f"Accuracy Score : {accuracy_score1}")
```

```
Accuracy Score : 0.5552065716356608
```

```python
# Print the classification report for the model
print("Classification Report")
print(classification_report(y_test, testing_predictions))
```

```
Classification Report
              precision    recall  f1-score   support

           0       0.88      0.56      0.69      7219
           1       0.14      0.50      0.22      1059

    accuracy                           0.56      8278
   macro avg       0.51      0.53      0.46      8278
weighted avg       0.79      0.56      0.63      8278
```

The accuracy score greatly reduced to 56%

```
Confusion Matrix
```

|          | Predicted 0 | Predicted 1 |
|----------|-------------|-------------|
| Actual 0 | 4068        | 3151        |
| Actual 1 | 531         | 528         |

```
Accuracy Score : 0.5552065716356608
```

### 4.1.3.3.3    Auto Optimization using Hyperparameter tuning¶

```python
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100],
    'class_weight': ["balanced"],
    'tol': [0.000001, 0.000005, 0.00001, 0.0001, 0.001],
    'penalty':["l2", "none"],
    'solver':["lbfgs", "newton-cg", "saga"],
    'max_iter': [50, 100, 150, 200, 550]
}
param_grid
```

```
{'C': [0.001, 0.01, 0.1, 1, 10, 100],
 'class_weight': ['balanced'],
 'tol': [1e-06, 5e-06, 1e-05, 0.0001, 0.001],
 'penalty': ['l2', 'none'],
 'solver': ['lbfgs', 'newton-cg', 'saga'],
 'max_iter': [50, 100, 150, 200, 550]}
```

```
# List the best parameters for this dataset
print(grid_clf.best_params_)
```

{'C': 0.01, 'class_weight': 'balanced', 'max_iter': 50, 'penalty': 'l2', 'solver': 'saga', 'tol': 0.001}

```
# List the best score
print(grid_clf.best_score_)
```

0.5467947170617254

```
# Make predictions with the hypertuned model
predictions = grid_clf.predict(X_test_scaled)
predictions
```

array([0, 0, 1, ..., 0, 0, 0], dtype=int64)

```
# Score the hypertuned model on the test dataset
grid_clf.score(X_test_scaled, y_test)
```

0.5565353950229524

```
# Calculate the classification report
print(classification_report(y_test, predictions,
                            target_names=["0", "1"]))
```

|              | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| 0            | 0.89      | 0.56   | 0.69     | 7219    |
| 1            | 0.14      | 0.50   | 0.23     | 1059    |
| accuracy     |           |        | 0.56     | 8278    |
| macro avg    | 0.52      | 0.53   | 0.46     | 8278    |
| weighted avg | 0.79      | 0.56   | 0.63     | 8278    |

Confusion Matrix

|          | Predicted 0 | Predicted 1 |
|----------|-------------|-------------|
| Actual 0 | 4074        | 3145        |
| Actual 1 | 526         | 533         |

Accuracy Score : 0.5565353950229524

## 4.1.3.4    Neural Network

### 4.1.3.4.1    Training with no Optimization

Preoptimization parameters

### Preoptimization

```
# Define the model - deep neural net, i.e., the number of input features and hidden nodes for each layer.
number_input_features = len(X_train_scaled[0])
hidden_nodes_layer1 =  20
hidden_nodes_layer2 =  10
outer_layer = 1

# Define the deep learning model
nn_model = tf.keras.models.Sequential()

# First hidden layer
nn_model.add(tf.keras.layers.Dense(units=hidden_nodes_layer1, activation="relu", input_dim=number_input_features))

# Second hidden layer
nn_model.add(tf.keras.layers.Dense(units=hidden_nodes_layer2, activation="relu"))

# Output layer
nn_model.add(tf.keras.layers.Dense(units=outer_layer, activation="sigmoid"))

# Check the structure of the model
print(nn_model.summary())
```

```
# Compile the Sequential model together and customize metrics
nn_model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])
```

```
# Train the model
fit_model = nn_model.fit(X_train_scaled, y_train, epochs=50)
```

```
Epoch 1/50
776/776 [==============================] - 1s 971us/step - loss: 0.4226 - accuracy: 0.8513
```

```
# Evaluate the model using the test data
nn_model_loss, nn_model_accuracy = nn_model.evaluate(X_test_scaled, y_test, verbose=2)
print(f"Loss: {nn_model_loss}, Accuracy: {nn_model_accuracy}")
```

```
259/259 - 0s - loss: 0.3750 - accuracy: 0.8744 - 351ms/epoch - 1ms/step
Loss: 0.3749982416629791, Accuracy: 0.8743658065795898
```

## 4.1.3.4.2  Auto Optimization using Hyperparameter tuning¶

**Auto Optimization using Hyperparameter tuning**

```
# Create a method that creates a new Sequential model with hyperparameter options
def create_model(hp):
    nn_model = tf.keras.models.Sequential()

    # Allow kerastuner to decide which activation function to use in hidden layers
    activation = hp.Choice('activation',['relu','tanh','sigmoid'])

    # Allow kerastuner to decide number of neurons in first layer
    nn_model.add(tf.keras.layers.Dense(units=hp.Int('first_units',
        min_value=1,
        max_value=10,
        step=2), activation=activation, input_dim=len(X_train_scaled[0])))

    # Allow kerastuner to decide number of hidden layers and neurons in hidden layers
    for i in range(hp.Int('num_layers', 1, 6)):
```

```
# Run the kerastuner search for best hyperparameters
tuner.search(X_train_scaled,y_train,epochs=20,validation_data=(X_test_scaled,y_test))
```

```
Trial 60 Complete [00h 01m 10s]
val_accuracy: 0.8741241693496704

Best val_accuracy So Far: 0.8750905990600586
Total elapsed time: 00h 17m 16s
INFO:tensorflow:Oracle triggered exit
```

```
# Get best model hyperparameters
best_hyper = tuner.get_best_hyperparameters(1)[0]
best_hyper.values
```

```
{'activation': 'tanh',
 'first_units': 9,
 'num_layers': 6,
 'units_0': 3,
 'units_1': 9,
 'units_2': 5,
 'units_3': 3,
 'units_4': 9,
 'units_5': 7,
 'tuner/epochs': 20,
 'tuner/initial_epoch': 0,
 'tuner/bracket': 0,
 'tuner/round': 0}
```

```
best_model = tuner.get_best_models(1)[0]
model_loss, model_accuracy = best_model.evaluate(X_test_scaled,y_test,verbose=2)
print(f"Loss: {model_loss}, Accuracy: {model_accuracy}")
```

```
259/259 - 2s - loss: 0.3752 - accuracy: 0.8751 - 2s/epoch - 6ms/step
Loss: 0.37516242265701294, Accuracy: 0.8750905990600586
```

# 5. Summary

These models were considered:

1. Random Forest
2. Decision Tree
3. Logistic Regression
4. Neural Network

| Model | Accuracy | |
|---|---|---|
| | Preoptimized | Optimized |
| Random Forest | 88% | 92% |
| Decision Tree | 87% | 90% |
| Logistic Regression | 56% | 56% |
| Neural Network | 87% | 88% |

Looking at the results of the models, Random Forest with the highest accuracy of 92% after optimization is clearly the model of choice for further application. It also has a 91% precision which is ideal because since we are looking at financial data, a higher precision avoids a lot of false positives saving the institution a lot. There will be less risk of approving a credit card for an applicant with bad credit.