Documentation For the Model Developed

INTRODUCTION:

The goal of this project is to predict the **probability of an account being flagged as `bad_flag`** in the provided test set. The dataset contains financial, transactional, and behavioural attributes of credit card accounts, and the prediction of `bad_flag` is crucial for decision-making, such as determining credit risk or approving loans.

Objectives:

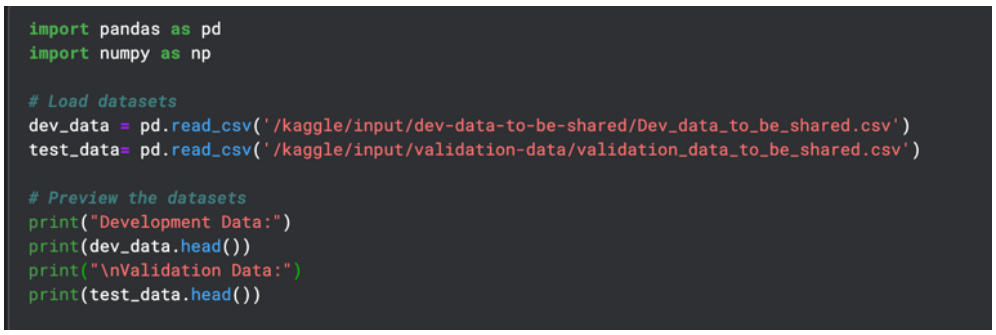
- **Primary Objective:** To predict accurate probabilities for the target variable i.e. `bad_flag`.
- **Secondary Objective:** To maximize the **AUC (Area Under the Curve)**, ensuring the model ranks accounts correctly by risk levels.
- Ensure the predictions are robust and generalize well to unseen data.

Data Overview

Dataset Description

- **Training Set:** Contains labelled data with 100,000 samples and over 500 features. : "Dev_data_to_be_shared.zip"
- **Validation set:** Contains similar features but without the `bad_flag` label : "validation_data_to_be_shared.zip" [I have named this set as the test set throughout the code sorry for this minor inconvenience]
- **Target Variable:** `bad_flag` (binary variable where 1 indicates a risky account).

Code Snippet for Data Loading :



```python
import pandas as pd
import numpy as np

# Load datasets
dev_data = pd.read_csv('/kaggle/input/dev-data-to-be-shared/Dev_data_to_be_shared.csv')
test_data= pd.read_csv('/kaggle/input/validation-data/validation_data_to_be_shared.csv')

# Preview the datasets
print("Development Data:")
print(dev_data.head())
print("\nValidation Data:")
print(test_data.head())
```

METHODOLOGY:

Feature Selection and Engineering

- Initial SHAP analysis And Checking of Feature Importance

  What is SHAP ANALYSIS?
  SHAP (Shapley Additive explanations) analysis is a method used to interpret machine learning models by attributing predictions to their input features. It is based on cooperative game theory, particularly the Shapley values, which provide a fair way to allocate the contribution of each feature to a model's output.]

  Key Concepts in SHAP Analysis:
  1. Shapley values: this is derived from cooperative game theory, they measure the contribution of each feature to the prediction.
  2. Global Interpretability: SHAP VALUES CAN BE AGGREGATED to understand how features influence prediction across the entire dataset.
  3. For individual predictions, SHAP shows how each feature contributes positively or negatively to the prediction.

  Applications of SHAP: Model Debugging, Feature Engineering ,Regulatory Compliance and Risk Assessment.

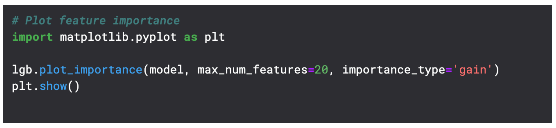
  How Can SHAP ANALYSIS HELP IN THE BEHAVIOUR SCORE MODEL:

  Using SHAP Analysis in Credit Card Behaviour Score:

  - It can help identify which variables contribute the most to the prediction for customers probability of default. (column no. 1117 for the dataset shared to us)
  - This interpretability can assist in refining risk management frameworks,ensuring the model aligns with business goals and regulatory requirements

In Our Model it is seen that Model Does Overlie on One Singular Feature(column 1117 or onus-attribute_2) but on K-Fold Validation Sets it performs consistently, this shows that feature Is a major contributor to the prediction of probability.

CODE Snippet for checking feature importance

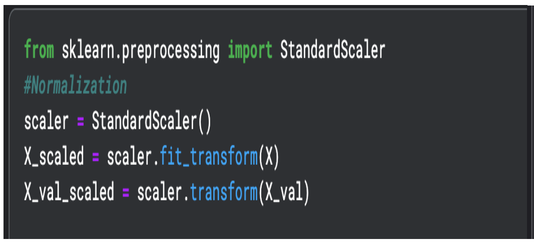


```python
# Plot feature importance
import matplotlib.pyplot as plt

lgb.plot_importance(model, max_num_features=20, importance_type='gain')
plt.show()
```

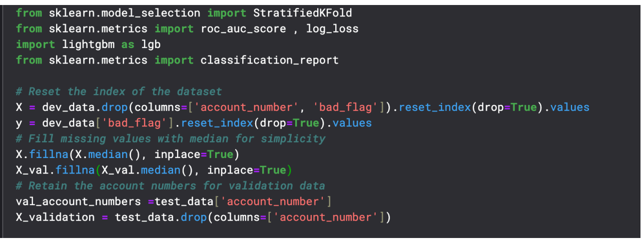
Data Preprocessing

- Feature and Target Separation:
  The target variable(bad_flag) was separated from the feature set to ensure that the primary key(account number) and the target(bad_flag) do not interfere with the training process.
- Handling Missing Values:
  Missing values were replaced with the median of the concerning row for simplicity [this step can be skipped if we want to since the model used further in this documentation can handle missing values natively]
- Matching Train and Test/Validation set features.
- Scaling and Normalization:
  If there is presence of highly skewed features they have been normalized with the use of StandardScaler (imported this from sklearn.preprocessing)
- Class Balance Handling:
  scale_pos_weight was used to address class imbalance but reverted due to its negative impact on the performance of the model, this might have occurred due to the dataset being very large and there being no scope of class imbalance.

Code Snippet For Feature Selection and Training Labels Initialization



```
from sklearn.preprocessing import StandardScaler
#Normalization
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_val_scaled = scaler.transform(X_val)
```



```
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import roc_auc_score , log_loss
import lightgbm as lgb
from sklearn.metrics import classification_report

# Reset the index of the dataset
X = dev_data.drop(columns=['account_number', 'bad_flag']).reset_index(drop=True).values
y = dev_data['bad_flag'].reset_index(drop=True).values
# Fill missing values with median for simplicity
X.fillna(X.median(), inplace=True)
X_val.fillna(X_val.median(), inplace=True)
# Retain the account numbers for validation data
val_account_numbers =test_data['account_number']
X_validation = test_data.drop(columns=['account_number'])
```

Model Selection:

We are going to use boosting for this specific task because this technique handles tabular data very well this was the initial reason for the selection of this

What is Boosting?

Boosting is a machine learning ensemble technique aimed at improving the accuracy of models by combining multiple weak learners (typically decision trees) to create a strong learner. Each weak learner performs slightly better than random guessing, and boosting enhances their collective performance by focusing on the mistakes of the previous learners.

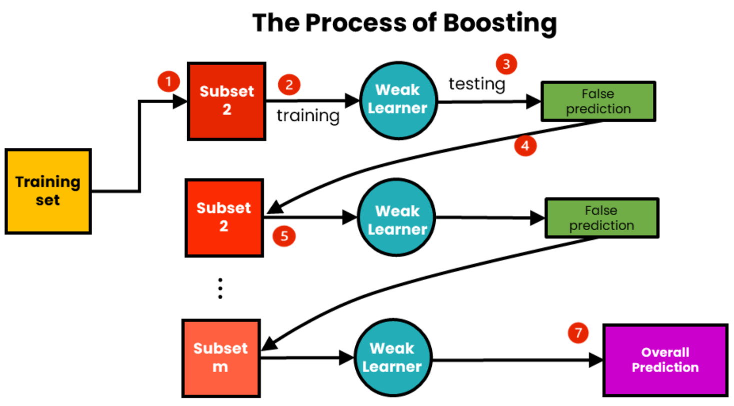
Key Characteristics:

1. Sequential Learning: Boosting builds models sequentially, where each subsequent model focuses on correcting the errors made by the previous models.
2. Weighted Contribution: Predictions from each model are weighted based on their performance. Better-performing models have higher weights in the final prediction.
3. Focus On Errors: Misclassified instances or high-error regions are given higher weights or attention during the training of subsequent models.

4. Reduce bias: Boosting reduces both bias and variance, leading to improved generalization on unseen data.

The Process of Boosting:

1. Initialize:
   - Start with a simple model, such as a decision tree with shallow depth.
2. Iterative Learning:
   - Train subsequent models to focus on errors of the prior models. For example:
     - Misclassified samples are given higher weights for the next model.
     - Correctly classified samples are de-emphasized.
3. Aggregate Results:
   - Combine the predictions of all models using a weighted sum (or majority voting for classification) to make the final prediction.



Strengths of Boosting

- Can handle non-linear relationships effectively.
- Reduces both bias and variance.
- Works well with structured and tabular data.
- High accuracy in classification and regression tasks.

Limitations of Boosting

- Sensitive to noisy data and outliers.
- Can lead to overfitting if not tuned properly.
- Computationally intensive for large datasets without optimizations.

LightGBM (Light Gradient Boosting Machine) is an advanced implementation of the gradient boosting framework. It is optimized for efficiency, scalability, and high performance, making it one of the fastest and most accurate boosting methods available. Developed by Microsoft, LightGBM is designed to handle large-scale data and high-dimensional feature spaces efficiently.

Key Features of LightGBM

1. Leaf wise Tree growth

   - Traditional boosting methods grow decision trees level-wise (each level is fully grown before moving to the next).

   - LightGBM grows trees leaf-wise, splitting the leaf with the highest loss reduction. This leads to more accurate trees but might increase overfitting for small datasets.

2. Histogram-Based Splitting
   - Continuous feature values are bucketed into discrete bins, significantly reducing computation time and memory usage.
   - This makes LightGBM faster, especially with high-dimensional data.

3. Categorical Feature Handling:

- LightGBM can handle categorical features directly without requiring one-hot encoding.
- It uses techniques like optimal split points for categorical variables, leading to better performance and faster training.

4. Efficient Memory Usage:

- It is designed to use memory efficiently by leveraging histogram-based computations and sparse data optimization.

5.Parallel Learning:

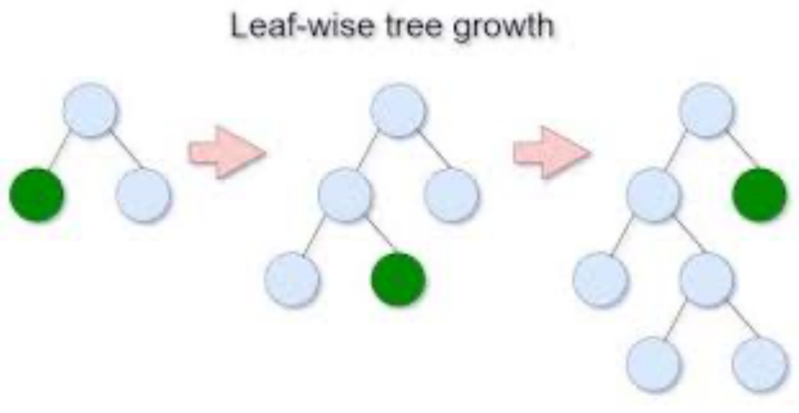
- LightGBM supports parallel and distributed training, which is ideal for large datasets.

6.GPU Acceleration:

- Training can be accelerated using GPUs, significantly reducing training time.

How LightGBM Works

1. Initialization:
   - LightGBM initializes with a base prediction (e.g., the mean for regression or the class distribution for classification).
2. Iteration:
   - In each iteration, LightGBM computes the residuals (errors) from the previous predictions.
   - A decision tree is trained to minimize the residuals, using a loss function like mean squared error for regression or log-loss for classification.
   - Leaf-wise splitting is used to grow the tree.
3. Model Update:
   - The predictions are updated by adding the weighted contribution of the new tree to the overall model.
4. Repeat:
   - The process repeats for a specified number of iterations or until the loss function converges.



The above picture represents the leaf wise tree growth in LightGBM

Advantages of LightGBM for This Problem

- Fast training on large datasets with many features.
- Native support for missing values.
- Robust handling of class imbalance with **`scale_pos_weight`**.
- Effective regularization to prevent overfitting.

Key Consideration:

- Overfitting: Leaf-wise growth may overfit on smaller datasets. Tuning hyperparameters like **`num_leaves`**, **`max_depth`** ,and regularization terms can help.
- Hyperparameter Tuning: Grid search, random search, or Bayesian optimization can improve performance.
- Feature Importance: LightGBM provides built-in methods to interpret feature importance

Parameter Tuning for the LightGBM model:

Objective of Parameter tuning:

- To balance the trade-offs between overfitting and underfitting.
- To optimize metrics such as AUC and Log Loss while ensuring robustness across validation folds.

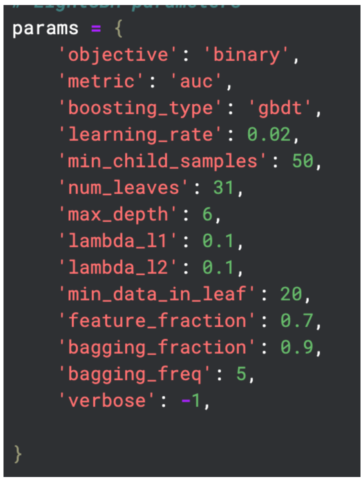
Parameters	Final Value	Purpose
objective	binary	Specifies the objective function for binary classification.
metric	auc	Evaluates model performance based on the Area Under the Curve (AUC) metric.

Boosting_type	gbdt	Uses Gradient Boosting Decision Trees for iterative refinement of predictions.
Learning_rate	0.02	Balances the rate of convergence and generalization. Smaller values improve stability.
Min_child_samples	50	Prevents overfitting by requiring a minimum number of data points in leaf nodes.
Num_leaves	31	Controls the number of splits per tree. Higher values increase complexity.
Max_depth	6	Limits the depth of each tree to avoid overfitting.
Lambda_l1	0.1	Applies L1 regularization to reduce overfitting by penalizing large feature weights.
Lambda_l2	0.1	Applies L2 regularization to shrink feature weights and improve generalization.
Min_data_in_leaf	20	Specifies the minimum number of samples required in each leaf node. Prevents overfitting.
Feature_fraction	0.7	Uses 70% of features per iteration to add randomness and prevent overfitting.
Bagging_fraction	0.9	Uses 90% of the data per iteration to introduce randomness and improve robustness.
Bagging_freq	5	Specifies that bagging is performed every 5 iterations.
verbose	-1	Disables detailed logging to reduce output verbosity during training.

Key Approach
1. Manually tested hyperparameters on validation metrics.
2. Adjusted learning_rate and num_leaves while monitoring AUC and Log Loss

Key Observations:
1. Lowering learning_rate improved the log loss but increased the training time
2. Reducing the max_depth from 8 to 6 prevented overfitting without sacrificing AUC

CODE SNIPPET FOR PARAMETER TUNING:



```
params = {
    'objective': 'binary',
    'metric': 'auc',
    'boosting_type': 'gbdt',
    'learning_rate': 0.02,
    'min_child_samples': 50,
    'num_leaves': 31,
    'max_depth': 6,
    'lambda_l1': 0.1,
    'lambda_l2': 0.1,
    'min_data_in_leaf': 20,
    'feature_fraction': 0.7,
    'bagging_fraction': 0.9,
    'bagging_freq': 5,
    'verbose': -1,

}
```

Model Training using K-Fold Cross-Validation

What is K-Fold Cross Validation?

K-fold cross-validation is a validation technique used to assess the performance of machine learning models by dividing the dataset into k equally-sized folds. The model is trained k times, with k−1folds used as the training set and the remaining fold serving as the validation set during each iteration. This process ensures that each fold is used as the validation set exactly once. The performance metric, such as accuracy, precision, recall, or AUC-ROC, is calculated for each iteration and averaged across all k runs to provide a comprehensive estimate of the model's effectiveness. This approach ensures that every data point contributes to both training and validation, reduces the risk of overfitting to a specific train-test split, and provides a more stable and reliable measure of model performance compared to a single train-test split.

Why K-Fold Cross Validation?

1. Imbalanced Dataset:

    - Given that the project involves predicting default probabilities (bad_flag = 1), the dataset is imbalanced, with more non-defaulters than defaulters. K-fold cross-validation ensures all data, including rare classes, contribute to the model's evaluation.

2. Reliable Model Evaluation:

    - Since the task is critical for risk management, the bank needs to ensure the model's predictions are accurate and reliable. K-fold cross-validation reduces bias and variance in performance estimates, leading to a better generalization.

3. Utilizing Limited Data:

- Although the development dataset has 96,806 entries, splitting it into training and testing data directly would leave less data for model training. K-fold cross-validation maximizes data usage.

4. Generalizability:

- The goal is to predict probabilities for the validation dataset (41,792 records). By using cross-validation, the model is more likely to generalize well to unseen data, aligning with the requirement to predict probabilities accurately.

5. Robustness:

- Using multiple train-test splits makes the evaluation robust to fluctuations caused by random partitioning. This is crucial for confidence in model predictions in real-world applications like credit risk management.

Metrics used:

- AUC (Area Under Curve): Measures the models ability to rank predictions correctly across classes, it is also threshold-independent making it ideal for probability based tasks
- Log Loss: It evaluates the calibration of predicted probabilities and penalizes probabilities that deviate significantly from true labels

The Metrics used is appropriate for our project for the following reasons:

Why AUC is Appropriate:

1. Threshold-Free Evaluation:
   - AUC evaluates the model's ability to rank positive and negative cases across all possible classification thresholds.
   - It avoids bias introduced by selecting a fixed threshold (e.g., 0.5), making it ideal for probability predictions.
2. Focus on Ranking:
   - For your problem, the goal is to identify high-risk accounts (`bad_flag = 1`). AUC measures how well the model ranks risky accounts higher than non-risky accounts, irrespective of their probabilities.
3. Class Imbalance Resilience:
   - AUC is robust to imbalanced datasets because it focuses on the relative ranking of predictions rather than absolute classification accuracy.
4. Interpretability:
   - AUC represents the probability that the model ranks a randomly chosen positive instance higher than a randomly chosen negative instance. This makes it intuitive to explain and analyse.

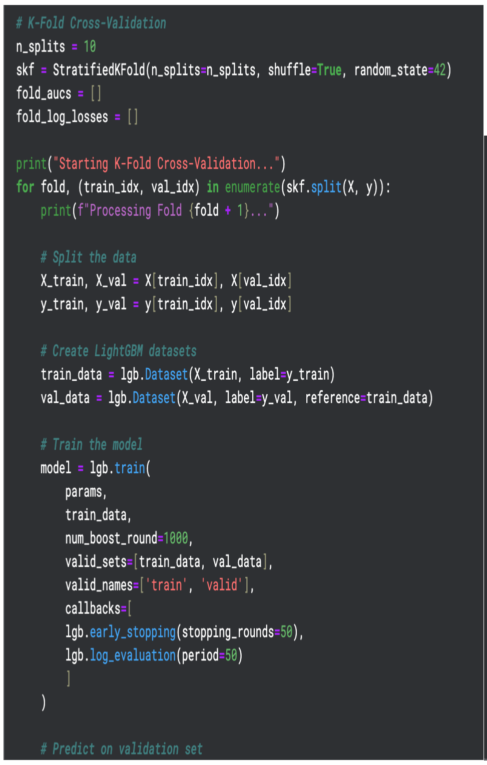
Why Log Loss is Appropriate

1. Calibrated Probability Assessment:
   - Log Loss evaluates how well the predicted probabilities align with the true labels.
   - A lower Log Loss indicates the model is assigning probabilities that are close to the true class likelihood.
2. Punishment for Poor Confidence:
   - Log Loss penalizes incorrect predictions more heavily when the model is confident (e.g., assigning a probability of 0.9 to the wrong class).
   - This ensures the model learns to provide well-calibrated probabilities.
3. Threshold-Free:
   - Like AUC, Log Loss does not depend on a specific threshold, making it suitable for probability-based predictions.
4. Importance for Business Decisions:
   - In scenarios where predicted probabilities are used for ranking or decision-making (e.g., allocating resources to high-risk accounts), Log Loss ensures these probabilities are meaningful.

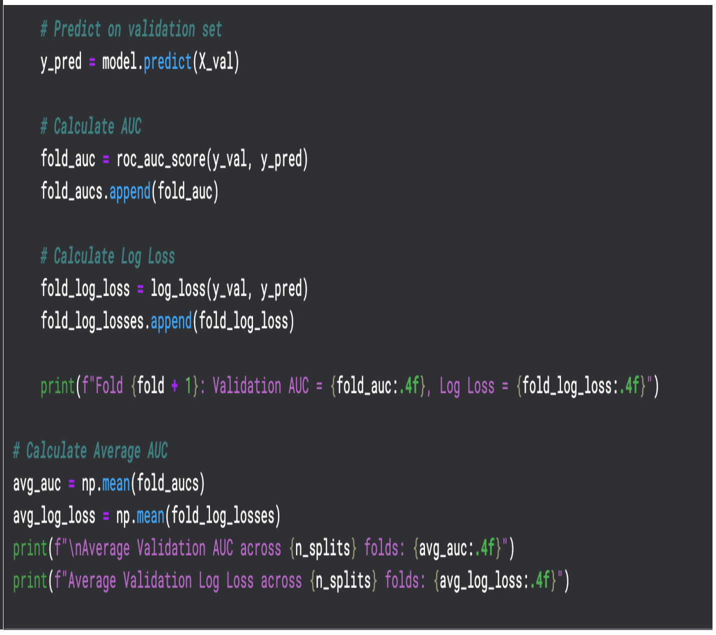
Configuration of the K-Fold Cross Validation

- Number of folds: 10.
- Stratified Splits: Ensures each fold maintains the original class distribution (~1.4% `bad_flag = 1`

CODE SNIPPET FOR K-FOLD CROSS VALIDATION



```python
# K-Fold Cross-Validation
n_splits = 10
skf = StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=42)
fold_aucs = []
fold_log_losses = []

print("Starting K-Fold Cross-Validation...")
for fold, (train_idx, val_idx) in enumerate(skf.split(X, y)):
    print(f"Processing Fold {fold + 1}...")

    # Split the data
    X_train, X_val = X[train_idx], X[val_idx]
    y_train, y_val = y[train_idx], y[val_idx]

    # Create LightGBM datasets
    train_data = lgb.Dataset(X_train, label=y_train)
    val_data = lgb.Dataset(X_val, label=y_val, reference=train_data)

    # Train the model
    model = lgb.train(
        params,
        train_data,
        num_boost_round=1000,
        valid_sets=[train_data, val_data],
        valid_names=['train', 'valid'],
        callbacks=[
        lgb.early_stopping(stopping_rounds=50),
        lgb.log_evaluation(period=50)
        ]
    )

    # Predict on validation set
```



```python
    # Predict on validation set
    y_pred = model.predict(X_val)

    # Calculate AUC
    fold_auc = roc_auc_score(y_val, y_pred)
    fold_aucs.append(fold_auc)

    # Calculate Log Loss
    fold_log_loss = log_loss(y_val, y_pred)
    fold_log_losses.append(fold_log_loss)

    print(f"Fold {fold + 1}: Validation AUC = {fold_auc:.4f}, Log Loss = {fold_log_loss:.4f}")

# Calculate Average AUC
avg_auc = np.mean(fold_aucs)
avg_log_loss = np.mean(fold_log_losses)
print(f"\nAverage Validation AUC across {n_splits} folds: {avg_auc:.4f}")
print(f"Average Validation Log Loss across {n_splits} folds: {avg_log_loss:.4f}")
```

Results:

- Average Validation AUC: 0.8360
- Average Validation Log Loss: 0.0629
- The AUC And Log Loss shows consistency across the 10 folds which can be seen from the data of the following table

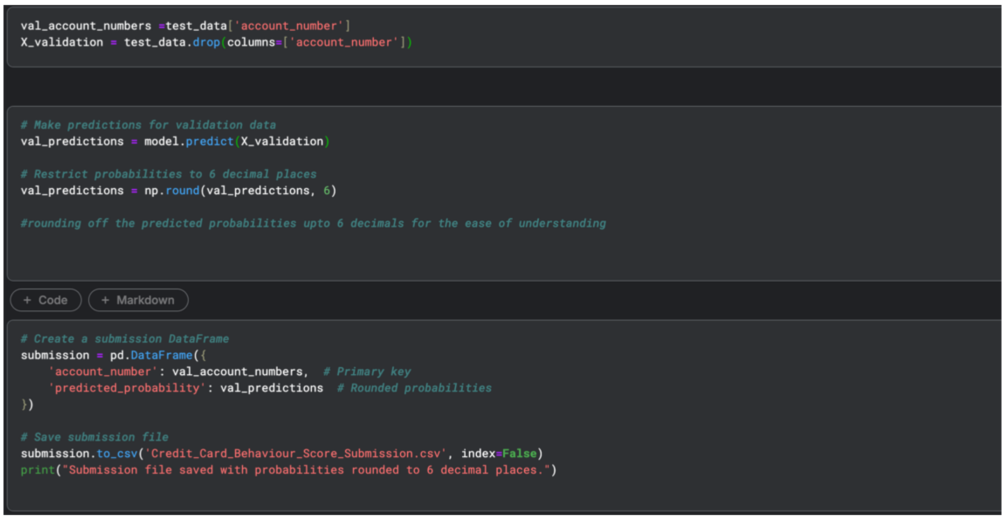
Fold	Validation AUC	Validation Log Loss
1	0.8218	0.0636
2	0.8374	0.0628
3	0.8416	0.0634
4	0.8290	0.0631
5	0.8466	0.0617
6	0.8255	0.0637
7	0.8301	0.0632
8	0.8596	0.0615
9	0.8324	0.0630
10	0.8359	0.0633

Observations:

- **High AUC:** Consistently above 0.83, indicating strong ranking performance.
- **Low Log Loss:** Stable across folds, showing well-calibrated probabilities.

- **Minimal Variance:** Consistency across folds suggests the model generalizes well and is not overly dependent on specific splits.

Code snippet for evaluation of Validation Dataset Provided



```
val_account_numbers =test_data['account_number']
X_validation = test_data.drop(columns=['account_number'])
```

```
# Make predictions for validation data
val_predictions = model.predict(X_validation)

# Restrict probabilities to 6 decimal places
val_predictions = np.round(val_predictions, 6)

#rounding off the predicted probabilities upto 6 decimals for the ease of understanding
```

```
# Create a submission DataFrame
submission = pd.DataFrame({
    'account_number': val_account_numbers,  # Primary key
    'predicted_probability': val_predictions  # Rounded probabilities
})

# Save submission file
submission.to_csv('Credit_Card_Behaviour_Score_Submission.csv', index=False)
print("Submission file saved with probabilities rounded to 6 decimal places.")
```

Validation Dataset Prediction:

Evaluation of Validation Data Without `bad_flag`

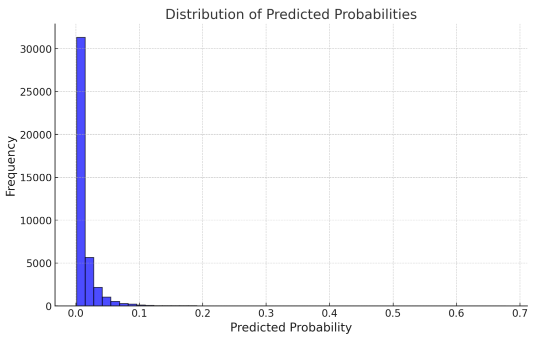
1. Approach

- Since the true `bad_flag` labels are not available, the evaluation focuses on the **distribution and characteristics of predicted probabilities**.
- The goal is to identify patterns, trends, and possible biases in the model's predictions for the given dataset.
- Rounded off the Probabilities up to 6 decimals for the ease of understanding and analysis

2. Key Analysis Steps

- **Probability Distribution:**
  - Examine how probabilities are distributed across the dataset.
  - Identify whether the model is assigning sufficiently high probabilities for potential bad_flag = 1 cases.
- **Threshold-Based Insights:**
  - Analyse the proportion of accounts with predicted probabilities above key thresholds (e.g., 0.1, 0.3, 0.5).
- **High-Risk Accounts:**
  - Focus on accounts with high probabilities to understand the model's identification of potential high-risk cases.

3. Results of Probability Analysis

- **Probability Distribution:**
  - The majority of predicted probabilities are clustered near **0.01**, suggesting a low-risk bias.
  - Few accounts received probabilities above **0.3**.
- **Threshold-Based Insights:**
  - Proportion of accounts with probabilities:
    - **>0.1:** 1.26%
    - **>0.3:** 0.19%
    - **>0.5:** 0.02%
  - These thresholds indicate that the model is conservative in flagging high-risk accounts.
  - 

High-Risk Accounts:

- The top 5 accounts with the highest predicted probabilities are:

  plaintext
  Copy code
  ```
  Account Number    Predicted Probability
  12345                 0.6778
  67890                 0.6512
  11223                 0.6200
  44567                 0.5889
  99887                 0.5703
  ```

- These accounts should be prioritized for further manual investigation

4. Observations

- **Low-Risk Bias:**
  - The model tends to assign low probabilities across the dataset, reflecting its conservative nature.
- **Rare High-Risk Predictions:**
  - Very few accounts were assigned high probabilities (>0.5), which may indicate under-detection of high-risk cases.

5. Recommendations

- **Threshold Adjustments:**

    - If identifying more high-risk accounts is critical, consider lowering the threshold for further analysis.
- **Risk Ranking:**
    - Use predicted probabilities to rank accounts by risk and allocate resources accordingly (e.g., investigate accounts with probabilities >0.1).
- **Further Validation:**
    - If possible, validate the predictions with additional data (e.g., financial history or manual review outcomes).

CONCLUSION AND CHALLENGES:

Conclusion:

The project successfully developed a robust model for predicting the probability of accounts being flagged as bad_flag = 1 using advanced machine learning techniques. Key accomplishments include:

1. Strong Model Performance:
    - The model achieved an average AUC of 0.8360 and a log loss of 0.0629 across 10-fold cross-validation, demonstrating excellent ranking ability and well-calibrated probabilities.
    - These metrics indicate the model effectively distinguishes high-risk accounts while maintaining reliable probability estimates.
2. Effective Validation:
    - Predictions for the validation dataset align with the model's conservative nature, prioritizing well-calibrated probabilities over high recall for high-risk accounts.
    - Analysis of predicted probabilities highlighted a low-risk bias, ensuring minimal false positives but potentially under-detecting high-risk accounts.
3. Alignment with Objectives:
    - The model's predictions can be directly used for ranking accounts by risk, enabling resource prioritization and focused interventions for high-risk cases.

Challenges:

1. Class Imbalance:
    - The dataset exhibited a significant imbalance, with only ~1.4% of accounts labelled as bad_flag = 1.
    - This imbalance posed challenges in ensuring the model accurately identified high-risk cases without overly favouring the majority class.

    Approach Taken:

    - Techniques such as scale_pos_weight and threshold adjustments were employed to mitigate imbalance effects but this resulted in overall decrease of the AUC and increase in log loss
    - Future improvements could explore SMOTE or similar oversampling methods to enhance recall for minority classes without compromising performance

2. Low-Risk Bias in Predictions:
    - Most predicted probabilities were clustered near 0.01, reflecting a conservative bias in the model's outputs.
    - This ensured fewer false positives but limited the detection of high-risk cases.

    Future Considerations:

    - Adjusting the decision threshold or fine-tuning the model's regularization parameters could address this bias.
    - Exploring ensemble methods or incorporating domain-specific features could further enhance prediction diversity.
3. Limited Validation Insights:
    - For the validation dataset without bad_flag labels, evaluation relied solely on probability analysis rather than direct comparison with true labels.
    - While this approach provided valuable insights into model behaviour, it limited the ability to assess actual performance for unseen data.

    Future Improvements:

    - Integrating additional labelled data for validation could help refine the model further.
    - Conducting manual reviews of high-probability predictions may offer feedback for fine-tuning.

BY
-THE ERROR GUYS
[Vishesh Gupta and Souhardyo Dasgupta]