Institute of Business Administration, Karachi

Financial Data Analytics - 98631

Final Project

A Machine Learning Approach to Detect Financial Fraud

Group Members
Tajwar Adil - 24836
Syed Muhammad Bilal - 24125
Shanzay Furqan - 24151
Sarosh Khalid - 24168
Ali Mehdi Shaikh - 24897

Course Instructor
Mr. Vishal Khemani

May 25th, 2025

Feature Engineering

```
# STEP 3: Feature Engineering
fraud_ids = set(fraudsters['USER_ID'])

txn_grp = transactions.groupby('USER_ID')
user_features = txn_grp.agg(
    trans_count=('ID', 'count'),
    total_amount=('AMOUNT_GBP', 'sum'),
    avg_amount=('AMOUNT_GBP', 'mean'),
    failed_count=('STATE', lambda x: (x == 'FAILED').sum()),
    currency_count=('CURRENCY', pd.Series.nunique)
).reset_index()
user_features['fail_rate'] = user_features['failed_count'] / user_features['trans_count']

type_dummies = pd.get_dummies(transactions['TYPE'])
type_counts = transactions[['USER_ID']].join(type_dummies).groupby('USER_ID').sum().reset_index()

features = pd.merge(user_features, type_counts, on='USER_ID', how='left')
features = pd.merge(features, users, left_on='USER_ID', right_on='ID', how='left')
today = datetime(2025, 5, 15)
features['CREATED_DATE'] = pd.to_datetime(features['CREATED_DATE'], errors='coerce')
features['BIRTH_DATE'] = pd.to_datetime(features['BIRTH_DATE'], errors='coerce')
if features['CREATED_DATE'].isnull().any() or features['BIRTH_DATE'].isnull().any():
    print("Warning: Some dates failed to parse. Check the original data format.")
features['account_age_days'] = (today - features['CREATED_DATE']).dt.days
features['user_age'] = ((today - features['BIRTH_DATE']).dt.days / 365).astype(int)

features.drop(['CREATED_DATE', 'BIRTH_DATE', 'ID'], axis=1, inplace=True)
features = pd.get_dummies(features, columns=['COUNTRY'], prefix='Country')
features['is_fraud'] = features['USER_ID'].isin(fraud_ids).astype(int)
features.drop('USER_ID', axis=1, inplace=True)

print("feature engineering completed")
```

Description:

In this project, several engineered features were created from the original users' and transactions' datasets to enhance the detection of fraudulent activities. At the user level, the total number of transactions **“trans_count”**, total transaction amounts **“total_amount”,** and average amount per transaction **“avg_amount”** were calculated to capture user activity levels and spending patterns. The number of failed transactions, **“failed_count,”** and the corresponding **“fail_rate”** were included to identify suspicious behaviors suggesting fraud attempts.

Additionally, the number of unique currencies used, **“currency_count,”** was engineered to highlight users engaging in complex or diverse transactions, which could raise fraud risks.

To represent transaction behaviors more specifically, one-hot encoding was applied to transaction types, with aggregated counts per user, offering insight into patterns such as frequent transfers or refunds. Time-based features such as **“account_age_days”,** measuring the time since account creation, and **“user_age”,** reflecting the user's age, were introduced to provide context around trustworthiness and potential anomalies.

Lastly, users were labeled as fraudulent or not **“is_fraud”** based on their presence in the fraudster’s dataset, establishing a target variable for supervised learning models. These engineered features

aim to summarize user behavior, transaction patterns, and time-based trends essential for building fraud detection models.

Machine Learning Models

1. Decision Tree

```
# STEP 5: Train Decision Tree Model
dt = DecisionTreeClassifier(
    max_depth=7,
    class_weight='balanced',  # handles imbalance
    random_state=42
)
dt.fit(X_train_scaled, y_train)

# STEP 6: Predictions and Evaluation
y_pred = dt.predict(X_test_scaled)
y_proba = dt.predict_proba(X_test_scaled)[:, 1]

print("=== Decision Tree Performance ===")
print(f"Accuracy:  {accuracy_score(y_test, y_pred):.3f}")
print(f"Precision: {precision_score(y_test, y_pred):.3f}")
print(f"Recall:    {recall_score(y_test, y_pred):.3f}")
print(f"F1 Score:  {f1_score(y_test, y_pred):.3f}")
print(f"ROC AUC:   {roc_auc_score(y_test, y_proba):.3f}")
```

Description:

This code trains and evaluates a Decision Tree model for fraud detection. The Decision Tree was chosen for its simplicity and ability to handle both numeric and categorical data. The **max_depth=7** limits tree complexity, helping prevent overfitting. To handle class imbalance, where fraudulent cases are much rarer than non-fraudulent cases, **class_weight='balanced'** ensures both classes are fairly treated during training. Setting **random_state=42** ensures reproducibility of results.

Standard scaling was applied to normalize feature values, so no feature dominates due to scale differences. After training, the model's performance is evaluated using metrics like accuracy, precision, recall, F1-score, and ROC AUC, providing insights into its effectiveness in identifying fraud.

2. K Means

```
# Save fraud label separately to compare after clustering
features['is_fraud'] = features['USER_ID'].isin(fraud_ids).astype(int)
fraud_labels = features['is_fraud']
features.drop(['USER_ID', 'is_fraud'], axis=1, inplace=True)

# STEP 4: Scaling
scaler = StandardScaler()
X_scaled = scaler.fit_transform(features)

# STEP 5: K-Means Clustering
n_clusters = 3  # You can adjust this based on your data
kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
clusters = kmeans.fit_predict(X_scaled)
features['cluster'] = clusters

# STEP 6: Evaluate Clustering
sil_score = silhouette_score(X_scaled, clusters)
print(f"Silhouette Score: {sil_score:.3f}")

# Visualize cluster distribution
sns.countplot(x=clusters)
plt.title("Cluster Counts (K-Means)")
plt.xlabel("Cluster")
plt.ylabel("Number of Users")
plt.show()
```

Description:

K-Means clustering is performed by the code to divide users into 3 different clusters **(n_clusters=3)**, chosen because of domain expertise or prior study. Ensuring zeroing in on the best result, the algorithm performs 10 random centroid initializations (**n_init=10)** and guarantees the same outcome by using **random_state=42**.

When using StandardScaler, we make the data's mean equal to zero and its variance equal to one for all features before clustering. Proper normalization is necessary so that the Euclidean distance metric does not favor features with range differences when K-Means makes clusters.

Points are assigned to clusters in the silhouette analysis, allowing us to measure exactly how each point fits better with its assigned cluster than with the others. Clusters are tightly grouped and distinct if the score is close to 1; when the score is near 0, clusters may merge. For a clear result, the silhouette score is always printed to three decimal places.

The size of each cluster is presented by placing the data points on a count plot, which can reveal which clusters hold far more fraudulent or non-fraudulent users.

3. KNN

```
# STEP 4: Train KNN Model
knn = KNeighborsClassifier(n_neighbors=5, metric='minkowski', p=2, n_jobs=-1)
knn.fit(X_train_scaled, y_train)

# STEP 5: Predictions and Evaluation
y_pred = knn.predict(X_test_scaled)
y_proba = knn.predict_proba(X_test_scaled)[:, 1]

print("=== KNN Performance ===")
print(f"Accuracy:  {accuracy_score(y_test, y_pred):.3f}")
print(f"Precision: {precision_score(y_test, y_pred):.3f}")
print(f"Recall:    {recall_score(y_test, y_pred):.3f}")
print(f"F1 Score:  {f1_score(y_test, y_pred):.3f}")
print(f"ROC AUC:   {roc_auc_score(y_test, y_proba):.3f}")
```

Description:

The model uses **n_neighbors=5** to consider the five nearest data points for classification, employing the Minkowski distance (with p=2, which equates to Euclidean distance) to measure proximity. The parameter **n_jobs=-1** allows the model to use all available processor cores for faster computation.

Before training, the data was scaled to standardize feature values, ensuring that distance calculations aren't skewed by varying feature scales. After training, the model's performance is evaluated using metrics like accuracy, precision, recall, F1-score, and ROC AUC, providing insights into its effectiveness in identifying fraud.

4. LightGBM

```
# STEP 6: Train LightGBM Model
lgb = LGBMClassifier(
    n_estimators=100,
    learning_rate=0.1,
    max_depth=7,
    random_state=42,
    class_weight='balanced'  # useful for handling imbalance
)
lgb.fit(X_train_scaled, y_train)

# STEP 7: Predictions and Evaluation
y_pred = lgb.predict(X_test_scaled)
y_proba = lgb.predict_proba(X_test_scaled)[:, 1]

print("=== LightGBM Performance ===")
print(f"Accuracy:  {accuracy_score(y_test, y_pred):.3f}")
print(f"Precision: {precision_score(y_test, y_pred):.3f}")
print(f"Recall:    {recall_score(y_test, y_pred):.3f}")
print(f"F1 Score:  {f1_score(y_test, y_pred):.3f}")
print(f"ROC AUC:   {roc_auc_score(y_test, y_proba):.3f}")
```

Description:

LightGBM is built on a gradient boosting framework, allowing it to handle classification and regression jobs both quickly and accurately. With the given configuration, 100 trees are built, and each new tree fixes a tenth of the errors from previous trees, guaranteeing fine performance without being made at an unnecessarily slow rate. No tree in the forest can be deeper than 7 (**max_depth=7)** so that fitting the data is not overdone and important nonlinearity in the data is found.

By default, LightGBM sets **class_weight='balanced',** which means the weights of minority class samples will be higher than the majority class when they are used less often. So, if the minority group covers only **5%** of the data, it will be treated and learned for training similarly to those that appear more often in the data.

This model is set up to take advantage of a standard feature set, which ensures the input data falls within the same range at all times and leads to more stable and accurate training. After processing data with LightGBM, users can control the classification results by adjusting the threshold value based on what is needed from their application. Thanks to its smooth implementation, MapReduce can process large datasets very quickly and use little memory. If you use a specific random seed, the program's output and model behavior will stay the same with each training.

5. Random Forest

```
# STEP 5: Train Random Forest Model
rf = RandomForestClassifier(
    n_estimators=100,
    max_depth=7,
    random_state=42,
    class_weight='balanced',  # handles imbalance
    n_jobs=-1
)
rf.fit(X_train_scaled, y_train)

# STEP 6: Predictions and Evaluation
y_pred = rf.predict(X_test_scaled)
y_proba = rf.predict_proba(X_test_scaled)[:, 1]

print("=== Random Forest Performance ===")
print(f"Accuracy:  {accuracy_score(y_test, y_pred):.3f}")
print(f"Precision: {precision_score(y_test, y_pred):.3f}")
print(f"Recall:    {recall_score(y_test, y_pred):.3f}")
print(f"F1 Score:  {f1_score(y_test, y_pred):.3f}")
print(f"ROC AUC:   {roc_auc_score(y_test, y_proba):.3f}")
```

Description:

Random Forest is used in many situations because it builds a lot of decision trees and brings together their decisions to improve **prediction accuracy** and decrease overfitting. To use the model, it makes 100 individual trees, where each tree has a maximum depth of 7, so it can pick up the most important data points without overfitting. The limitation allows the model to work well on data it hasn't seen.

Handling class imbalance is one of the most important problems in many classification problems. To deal with this, Random Forest includes the **class_weight='balanced'** option, which gives greater weight to rare or minority classes when learning. Because of this weighting, the model is better able to discover important but less common classes since they are assigned a higher impact on tree building.

When using this model, the **features are scaled** so that their ranges are equalized. Scaling the data ensures that the algorithm reaches a solution more smoothly and avoids partiality because of differences in feature scales. As soon as training is done, the model can be used to provide both categorical classifications and their corresponding **probabilities,** which opens the possibility of setting all kinds of thresholds.

Model accuracy is measured in detail using Accuracy, Precision, Recall, F1 Score, and ROC AUC. All of these measurements let us see clearly how well the Random Forest does, mainly when it needs to recognize rare class instances. Also, when using a fixed seed, the outcomes can be repeated, so it's easy to compare different models or conduct the same experiments again.

6. XGBoost

```
# STEP 4: Train XGBoost Model
xgb = XGBClassifier(
    n_estimators=100,
    learning_rate=0.1,
    max_depth=7,
    scale_pos_weight=(len(y_train)-sum(y_train))/sum(y_train),  # handles imbalance
    use_label_encoder=False,
    eval_metric='logloss',
    random_state=42
)
xgb.fit(X_train_scaled, y_train)

# STEP 5: Predictions and Evaluation
y_pred = xgb.predict(X_test_scaled)
y_proba = xgb.predict_proba(X_test_scaled)[:, 1]

print("=== XGBoost Performance ===")
print(f"Accuracy:  {accuracy_score(y_test, y_pred):.3f}")
print(f"Precision: {precision_score(y_test, y_pred):.3f}")
print(f"Recall:    {recall_score(y_test, y_pred):.3f}")
print(f"F1 Score:  {f1_score(y_test, y_pred):.3f}")
print(f"ROC AUC:   {roc_auc_score(y_test, y_proba):.3f}")
```

Description:

The model is configured **with n_estimators=100** to limit the number of boosting rounds and **learning_rate=0.1** to control how much each tree contributes to the overall prediction. The **max_depth=7** restricts tree depth to prevent overfitting.

Since fraud cases are rare, the model adjusts the **scale_pos_weight** based on the ratio of non-fraud to fraud cases, making sure it doesn't overlook fraud. The settings **use_label_encoder=False** and **eval_metric='logloss'** ensure smooth binary classification without outdated warnings

Standard scaling was applied to normalize feature values, so no feature dominates due to scale differences. After training, the model's performance is evaluated using metrics like accuracy, precision, recall, F1-score, and ROC AUC, providing insights into its effectiveness in identifying fraud.

7. Stacking: LightGBM + Random Forest

```
# STEP 5: Define Base Models with Adjusted Class Weights
lgb = LGBMClassifier(n_estimators=100, learning_rate=0.1, max_depth=7, class_weight={0:1, 1:5}, random_state=42)
rf = RandomForestClassifier(n_estimators=100, max_depth=7, class_weight={0:1, 1:5}, random_state=42, n_jobs=-1)

# STEP 6: Define Meta Model
meta_model = LogisticRegression(solver='lbfgs', max_iter=1000)

# STEP 7: Create and Train Stacking Classifier
stacking_model = StackingClassifier(
    estimators=[('lgb', lgb), ('rf', rf)],
    final_estimator=meta_model,
    passthrough=False,
    cv=5,
    n_jobs=-1
)

stacking_model.fit(X_train_resampled, y_train_resampled)

# STEP 8: Evaluation with Adjusted Threshold
y_proba = stacking_model.predict_proba(X_test_scaled)[:, 1]
threshold = 0.3  # Adjust as needed
y_pred = (y_proba >= threshold).astype(int)

print("=== Stacking Model (LightGBM + Random Forest) Performance ===")
print(f"Accuracy:  {accuracy_score(y_test, y_pred):.3f}")
print(f"Precision: {precision_score(y_test, y_pred, zero_division=0):.3f}")
print(f"Recall:    {recall_score(y_test, y_pred):.3f}")
print(f"F1 Score:  {f1_score(y_test, y_pred):.3f}")
print(f"ROC AUC:   {roc_auc_score(y_test, y_proba):.3f}")
```

Description:

This code builds a stacking model combining LightGBM and Random Forest classifiers with Logistic Regression as the meta-learner. The LightGBM and Random Forest base models are configured with **n_estimators=100**, **max_depth=7**, and **class_weight** adjustments to address class imbalance to ensure fair treatment of fraud and non-fraud cases.

The meta-learner, Logistic Regression, is used for its simplicity and effectiveness in blending model outputs into final predictions. The stacking classifier is created using **cross-validation (cv=5)** and parallel computation **(n_jobs=-1)** for efficiency.

Standard scaling was applied to normalize feature values, so no feature dominates due to scale differences. After training, the model's performance is evaluated using metrics like accuracy, precision, recall, F1-score, and ROC AUC, providing insights into its effectiveness in identifying fraud.

8. Stacking: LightGBM + XGBoost

```
# STEP 5: Define Base Models with Adjusted Class Weights
lgb = LGBMClassifier(n_estimators=100, learning_rate=0.1, max_depth=7, class_weight='balanced', random_state=42)
xgb = XGBClassifier(n_estimators=100, learning_rate=0.1, max_depth=7, scale_pos_weight=(len(y_train)-sum(y_train))/s

# STEP 6: Define Meta Model
meta_model = LogisticRegression(solver='lbfgs', max_iter=1000)

# STEP 7: Create and Train Stacking Classifier
stacking_model = StackingClassifier(
    estimators=[('lgb', lgb), ('xgb', xgb)],
    final_estimator=meta_model,
    passthrough=False,
    cv=5,
    n_jobs=-1
)

stacking_model.fit(X_train_scaled, y_train_resampled)

# STEP 8: Evaluation with Adjusted Threshold
y_proba = stacking_model.predict_proba(X_test_scaled)[:, 1]
threshold = 0.3  # Adjust as needed
y_pred = (y_proba >= threshold).astype(int)

print("=== Stacking Model (LightGBM + XGBoost) Performance ===")
print(f"Accuracy:  {accuracy_score(y_test, y_pred):.3f}")
print(f"Precision: {precision_score(y_test, y_pred, zero_division=0):.3f}")
print(f"Recall:    {recall_score(y_test, y_pred):.3f}")
print(f"F1 Score:  {f1_score(y_test, y_pred):.3f}")
print(f"ROC AUC:   {roc_auc_score(y_test, y_proba):.3f}")
```

Description:

To create this model, the key gradient boosting classifiers **LightGBM** and **XGBoost** are used and they are each tuned with 100 estimators, a learning rate of 0.1 and a tree depth limit of 7. They decide how simple or complex the model will be during its training. To correct for the imbalance in the data, class weights are modified, resulting in better detection of the rare class.

A meta-model is built using Logistic Regression, with the '**lbfgs' solver** and allowing up to 1000 iterations for base model predictions. Because of its design, the meta-learner can combine the advantages of LightGBM and XGBoost to provide more correct predictions. The training process uses **5 subsets,** and the stacking classifier runs on multiple CPU cores for fast processing.

To evaluate our predictions, the probabilities are converted into **class labels** only if they are above **0.3.** This adjustment enables the model to better weigh false positives against false negatives, which boosts how it works in the real world. The model shows how well it works with the help of accuracy, precision, recall, F1 score and ROC AUC which reflect both how accurately it categorizes and how it differentiates between classes.

9. Stacking: XGBoost and KNN

```
# STEP 4: Define Base Models
xgb = XGBClassifier(n_estimators=100, learning_rate=0.1, max_depth=7, scale_pos_weight=(len(y_train)-sum(y_train))/su
                    use_label_encoder=False, eval_metric='logloss', random_state=42)
knn = KNeighborsClassifier(n_neighbors=5, metric='minkowski', p=2, n_jobs=-1)

# STEP 5: Define Meta Model
meta_model = LogisticRegression(solver='lbfgs', max_iter=1000)

# STEP 6: Create and Train Stacking Classifier
stacking_model = StackingClassifier(
    estimators=[('xgb', xgb), ('knn', knn)],
    final_estimator=meta_model,
    passthrough=False,
    cv=5,
    n_jobs=-1
)

stacking_model.fit(X_train_scaled, y_train)

# STEP 7: Evaluation
y_pred = stacking_model.predict(X_test_scaled)
y_proba = stacking_model.predict_proba(X_test_scaled)[:, 1]

print("=== Stacking Model (XGBoost + KNN) Performance ===")
print(f"Accuracy:  {accuracy_score(y_test, y_pred):.3f}")
print(f"Precision: {precision_score(y_test, y_pred):.3f}")
print(f"Recall:    {recall_score(y_test, y_pred):.3f}")
print(f"F1 Score:  {f1_score(y_test, y_pred):.3f}")
print(f"ROC AUC:   {roc_auc_score(y_test, y_proba):.3f}")
```

Description:

Stacking ensemble learning is used in this model which combines the strengths of two leading base classifiers: **XGBoost** and **K-Nearest Neighbors (KNN).** It has been tuned with 100 estimators, **a learning rate of 0.1** and a max depth of 7 to achieve good performance and speed with early stopping and log-loss evaluation. A KNN model is constructed with 5 neighbors and measures using the **Minkowski distance**. Since each base model focuses on something different in the data, ensembles bring together multiple useful patterns in their predictions.

When stacking, the chosen meta model is **a Logistic Regression classifier** which balances the predictions from the base models. Stacking is trained by using 5-fold cross-validation for strong and general results. To evaluate final performance, we measure the model on the test set using metrics like Accuracy, Precision, Recall, F1 Score and ROC AUC which together show how accurately the model finds true positives and how well it prevents false positives.

Following this combination, the model achieves a high level of accuracy as well as dependable performance on classification tasks. With stacking, multiple learning perspectives are sensibly combined in the meta-model, allowing it to perform far better than its **individual classifiers**.