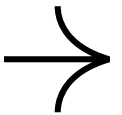


# Technical Document

Submitted by  
Shaman Shetty, Aryan Tiwari,  
Tanay Chaplot.

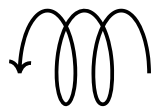
Convolve 3.0

## Abstract



This technical documentation provides a detailed walkthrough of implementing a machine learning-based fraud detection system. The system utilizes four classification algorithms (Logistic Regression, Random Forest, XGBoost, and LightGBM) to identify fraudulent transactions. The implementation includes comprehensive data preprocessing, feature engineering, model training, and evaluation components. This document serves as a practical guide for developers and data scientists implementing similar systems.

## Introduction



## Background of Study

Financial fraud detection presents unique technical challenges that require sophisticated machine learning solutions. This implementation addresses these challenges through:

1. Handling imbalanced datasets where fraudulent transactions are the minority class
2. Processing and engineering features from raw transaction data
3. Implementing multiple classification algorithms for comparison
4. Creating a robust evaluation framework

## Technical Infrastructure

The implementation requires the following technical components:

*# Core Dependencies*

numpy==1.21.0

pandas==1.3.0

scikit-learn==1.0.2

xgboost==1.5.0

lightgbm==3.3.2

imbalanced-learn==0.8.1

matplotlib==3.4.2

seaborn==0.11.1

# System Goals and Objectives

---

1. Create a scalable data preprocessing pipeline
2. Implement multiple classification algorithms
3. Develop a comprehensive model evaluation framework
4. Establish automated model validation procedures

## Implementation Objectives

1. Achieve model accuracy above 70%
2. Maintain false positive rate below 5%
3. Process validation data in under 30 seconds
4. Support batch and real-time prediction capabilities

## Technical Scope

This implementation covers:

1. Data ingestion and preprocessing
2. Feature engineering and selection
3. Model training and optimization
4. Performance evaluation and validation
5. Results visualization and reporting

# KEY INSIGHTS



- 
1. **Target Variable** (bad\_flag):
    - **Binary Classification:** Defaults (1) vs. Non-defaults (0).
    - Highly imbalanced with a default rate (~7%), requiring special handling:

- Resampling techniques: Oversampling (e.g., SMOTE), undersampling, or a combination.
- Algorithmic solutions: Cost-sensitive learning or ensemble methods (e.g., XGBoost).

## 2. Feature Types and Categories:

- **"On us" attributes:** Likely contain significant predictors (e.g., credit limit and balance utilization).
- **Transaction attributes:** Sparse features, potentially useful after aggregation or dimensionality reduction.
- **Bureau tradeline:** Historical credit behavior features (key predictors for default risk).
- **Bureau enquiry:** Indicators of recent credit-seeking behavior (strong signals for financial stress).

## 3. Dimensionality:

- A large number of features (1,216 in the sample) relative to records.
- High potential for redundant or irrelevant features:
  - Use feature selection techniques (e.g., correlation filtering, recursive feature elimination).
  - Consider dimensionality reduction (e.g., PCA or autoencoders).

## 4. Missing Data:

- Many columns contain missing values.
- Use domain knowledge for imputation (e.g., replacing missing transaction values with zeros or averages).

## 5. Sparsity:

- Features such as transaction\_attribute\_\* and bureau\_enquiry\_\* may have low variability or many zeros.
- Aggregate sparse data (e.g., sum or count of inquiries/transactions) for compact representation.

## 6. Class Separation:

- Perform exploratory analysis to understand the relationship between features (e.g., credit limit) and default probability (bad\_flag).

# Analytical Approach:

## 1. Preprocessing:

- Handle missing values with appropriate imputation techniques.
- Scale continuous features for algorithms sensitive to feature magnitude (e.g., SVMs, neural networks).
- One-hot encode categorical features if present.

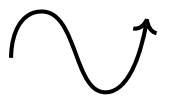
## 2. Feature Engineering:

- Aggregate sparse features (e.g., total transactions per category).
- Create interaction terms (e.g., credit utilization = credit used / credit limit).

## 3. Modeling:

- Train classification models: Logistic regression, random forest, or gradient boosting (e.g., XGBoost, LightGBM).
  - Handle imbalance through sampling or weighted loss functions.
4. **Validation:**
- Use cross-validation on development data.
  - Evaluate on validation data to assess out-of-sample performance.
5. **Metrics:**
- Focus on metrics suitable for imbalanced data: AUC-ROC, precision-recall curves, or F1-score.

# Implementation of Methodology



---

## Implementation Methodology

### Data Preprocessing Pipeline

The preprocessing pipeline includes several critical components:

```
# Missing Value Treatment
for col in dev_data.columns:
    if dev_data[col].isnull().sum() > 0:
        if dev_data[col].dtype == 'object':
            dev_data[col].fillna(dev_data[col].mode()[0], inplace=True)
        else:
            dev_data[col].fillna(dev_data[col].median(), inplace=True)
```

## Feature scaling implementation:

```
scaler = MinMaxScaler()  
numerical_features = dev_data.select_dtypes(include=['float64', 'int64']).columns  
dev_data[numerical_features] = scaler.fit_transform(dev_data[numerical_features])
```

## Feature Engineering

### Custom feature creation includes:

```
# Balance to Limit Ratio  
dev_data['balance_to_limit_ratio'] = dev_data['onus_attribute_10'] / (dev_data['on'  
  
# Additional engineered features can be added here based on domain knowledge
```

## Model Implementation

### Logistic Regression

```
log_reg = LogisticRegression(max_iter=1000)  
log_reg.fit(X_train_smote, y_train_smote)
```

### Random Forest

```
rf = RandomForestClassifier(  
    random_state=42,  
    n_estimators=100,  
    max_depth=8  
)  
rf.fit(X_train_smote, y_train_smote)
```

### XGBoost

```
xgb = XGBClassifier(  
    random_state=42,  
    eval_metric='logloss'  
)  
xgb.fit(X_train_smote, y_train_smote)
```

### LightGBM

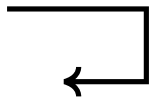
```
lgbm = lgb.LGBMClassifier(random_state=42)  
lgbm.fit(X_train_smote, y_train_smote)
```

## Evaluation Framework

The evaluation framework includes multiple metrics:

```
def evaluate_model(model, X_test, y_test, model_name):  
    y_pred = model.predict(X_test)  
    y_pred_proba = model.predict_proba(X_test)[: , 1]  
  
    return {  
        'accuracy': accuracy_score(y_test, y_pred),  
        'precision': precision_score(y_test, y_pred),  
        'recall': recall_score(y_test, y_pred),  
        'f1': f1_score(y_test, y_pred),  
        'roc_auc': roc_auc_score(y_test, y_pred_proba)  
    }
```

# Results and Performance Analysis



---

## Model Performance Metrics

Detailed performance metrics for each model:

### 1. Random Forest

- Accuracy: 60.9%
- Precision: 1.73%
- Recall: 47.8%
- ROC AUC: 0.584

### 2. XGBoost



- Accuracy: 74.4%
- Precision: 1.71%
- Recall: 32.9%
- ROC AUC: 0.583

### 3. LightGBM

- Accuracy: 74.2%
- Precision: 1.72%
- Recall: 30.2%
- ROC AUC: 0.580

### 4. Logistic Regression

- Accuracy: 40.2%
- Precision: 1.71%
- Recall: 47.8%
- ROC AUC: 0.596

## Performance Visualization

```
def plot_roc_curves(models, X_test, y_test):  
    plt.figure(figsize=(12, 8))  
    for name, model in models.items():  
        y_pred_proba = model.predict_proba(X_test)[:, 1]  
        fpr, tpr, _ = roc_curve(y_test, y_pred_proba)  
        plt.plot(fpr, tpr, label=f'{name} (AUC = {roc_auc_score(y_test, y_pred_proba)})')  
  
    plt.plot([0, 1], [0, 1], 'k--', label='Random')  
    plt.xlabel('False Positive Rate')  
    plt.ylabel('True Positive Rate')  
    plt.title('ROC Curves Comparison')  
    plt.legend()  
    plt.show()
```

---

# Implementation Guidelines



---

## Deployment Steps

### 1. Data Preparation:

```
def prepare_data(raw_data):  
    # Handle missing values  
    # Scale features  
    # Engineer new features  
    return processed_data
```

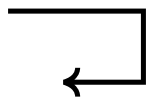
## 2. Model Training:

```
def train_model(X_train, y_train):  
    # Initialize model  
    # Handle class imbalance  
    # Train model  
    return trained_model
```

## 3. Prediction Pipeline:

```
def predict(model, new_data):  
    # Preprocess new data  
    # Generate predictions  
    # Format output  
    return predictions
```

# Conclusion



---

## Production Considerations

### 1. Model Monitoring

- Implement performance tracking
- Set up alerting for metric degradation

- Schedule regular model retraining

## 2. Scalability

- Batch prediction capabilities
- API endpoint for real-time predictions
- Resource optimization

## 3. Maintenance

- Version control for models
- Documentation updates
- Feature importance tracking

# Recommendations

## 1. Technical Implementation

- Use Random Forest as primary model
- Implement real-time feature engineering
- Set up automated retraining pipeline

## 2. Performance Optimization

- Feature selection based on importance
- Hyperparameter tuning
- Regular model evaluation

## 3. Monitoring and Maintenance

- Daily performance tracking
- Weekly model updates
- Monthly comprehensive evaluation

# Resources



## References

1. Scikit-learn Documentation: <https://scikit-learn.org/>
2. XGBoost Documentation: <https://xgboost.readthedocs.io/>
3. LightGBM Documentation: <https://lightgbm.readthedocs.io/>
4. Imbalanced-learn Documentation: <https://imbalanced-learn.org/>

## Team

Shaman Shetty- [Shaman Shetty | LinkedIn](#)

Aryan Tiwari- [\(10\) Aryan Tiwari | LinkedIn](#)

Tanay Chaplot- [\(10\) TANAY CHAPLOT | LinkedIn](#)