Credit Card Default Prediction Using Machine Learning

CS 677 Final Project – Presentation | December 18, 2025 | Fnu Ashutosh (U01955320), Atharva Pande (U01985210), Kenji Okura (U01769019)

PROBLEM STATEMENT

Statement: Predict whether a credit card customer will default on their next payment, which enables proactive risk management and minimizing financial losses.

WHY? – THE BUSINESS IMPACT

Accurate predictions are needed to minimize financial loss for the credit card company. On average:

- Not predicting a default (false negative) costs $10,000 per customer
- Incorrectly predicting a default (false positive) can cost $200 per customer
- Class imbalance is 77.88 non-default to 22.12 default (3.52:1 ratio).

Because of the high cost of defaults, recall (or catching the defaults) a critical metrics.

The model would need to prioritize catching true defaulters, while trying to minimize the chances of false positives. All to maximize profits.

PROBLEM

Dataset

DATASET

UC Irvine's Machine Learning Repository - Default of Credit Card Clients

QUICK STATS

- Sample Size: 30,000 customers
  - Training – test split: 80-20 (24,000 training, 6,000 test samples).
- Time period: April 2005 – September 2005 (6 months)
- Number of features: 23
  - All do not have null values (no missing data!)
  - All are integer data types
  - Example include sex, limit_bal, education, payment status, etc.
- The target variable (if they defaulted next month) is binary (0 or 1).

UC Irvine
Machine Learning
Repository

WHY THIS DATASET?

- Real-world relevance with a sufficient size and many features. The default rate balance of 22.12 reflects realw work distributions.

Exploratory Data Analysis (EDA)

The Dataset Structure, Basic Information, Statistics, and quick analysis

Shape and Size

30,000 samples
23 features
Totals to 690,000 data points

Data Types

The features are all integers
The target is a binary 0 or 1

Together they take 5.3 mb of space

Stats

We calculate the mean, standard deviation, min max etc. Some insights are:
- Credit Limit Range: NT$10,000 - NT$1,000,000
• Average Credit Limit: NT$167,484
• Age Range: 21 - 79 years
• Average Age: 35.5 years

Missing Data

The source of data indicated there is no missing data. Analysis agreed with this.

Duplicate Data

There are 35 duplicate data.

Outliers

We tested outliers on select features (like age). We discover that LIMIT_BAL is most effected.

Exploratory Data Analysis (EDA)

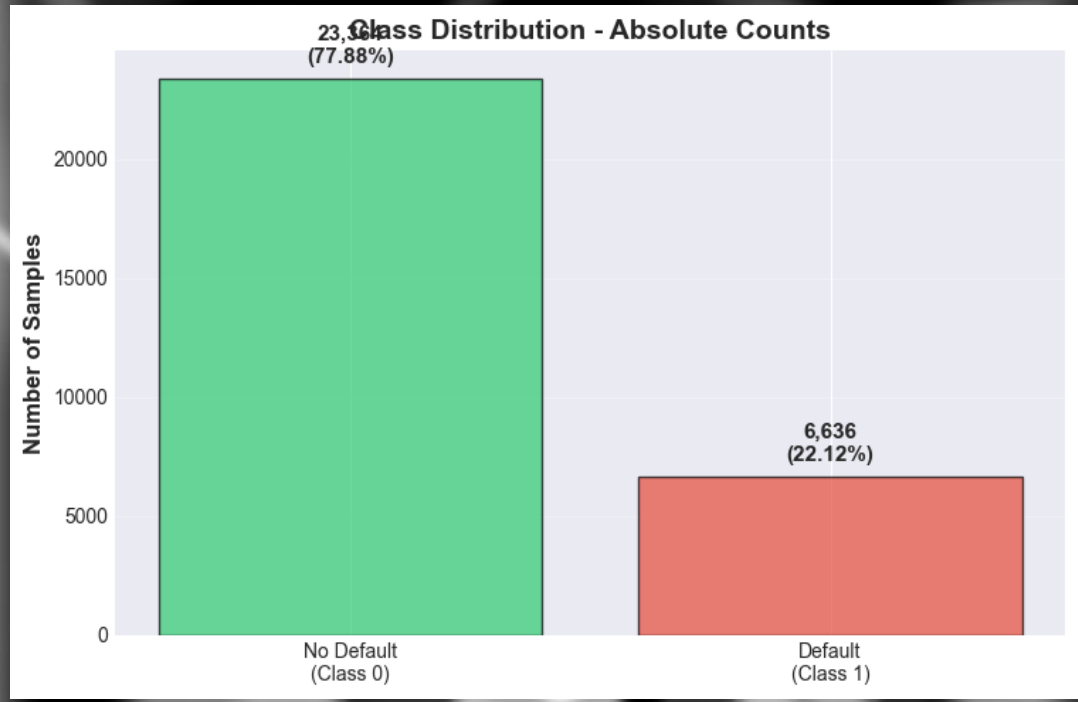
Target Variable Distribution

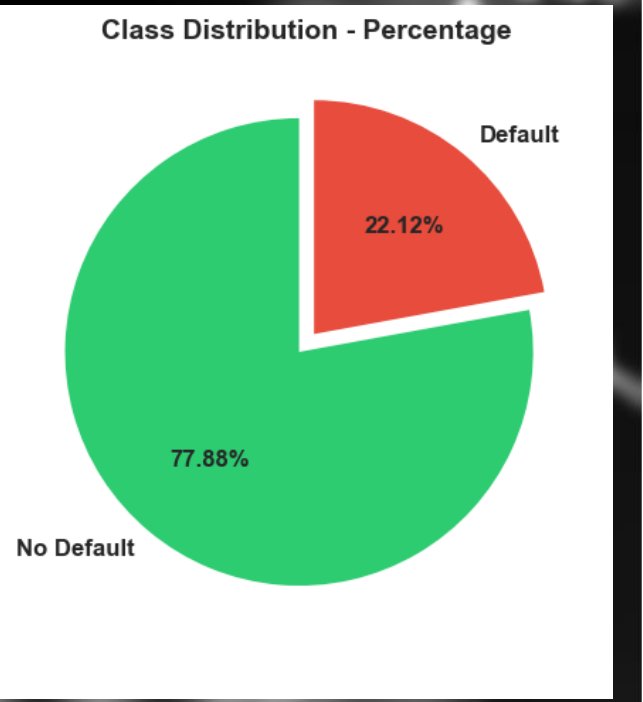
Class Distribution

Why?
To apply correct evaluation metrics, decide on sampling strategies or setting realistic expectation

What we learned?
Moderately imbalanced, meaning class aware techniques is needed.

More than just testing for accuracy is needed.





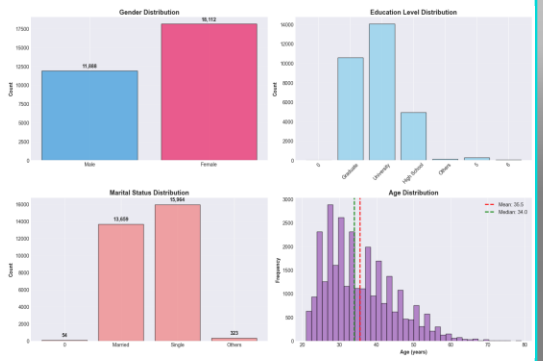
Exploratory Data Analysis (EDA)

Analysis on Features

Demographic Features

Ex. Sex, education. marriage status, age.

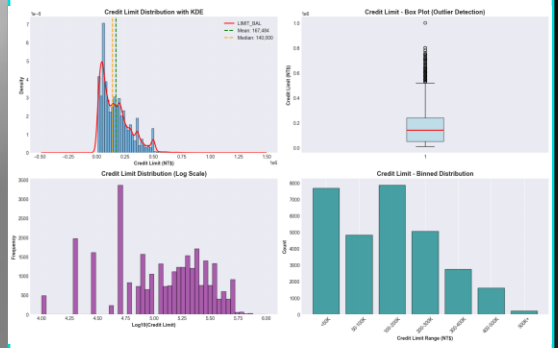
For this case we calculated ratios of them for example.



Credit Limit Analysis

Ex. LIMIT_BAL

Find distribution, patterns or outliers



Payment Status

Ex. PAY_0….PAY_6

When analyzed with theme, we learn that payment behavior is improving

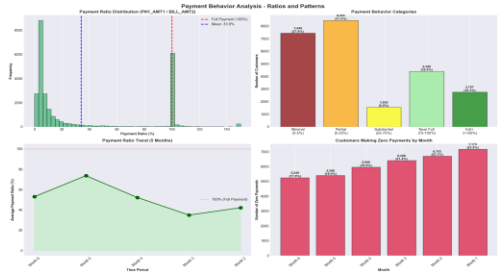
Bill Amount

Ex. BILL_AMT1
…. BILL_AMT6

Learned insights like high utilization is a strong default risk indicator

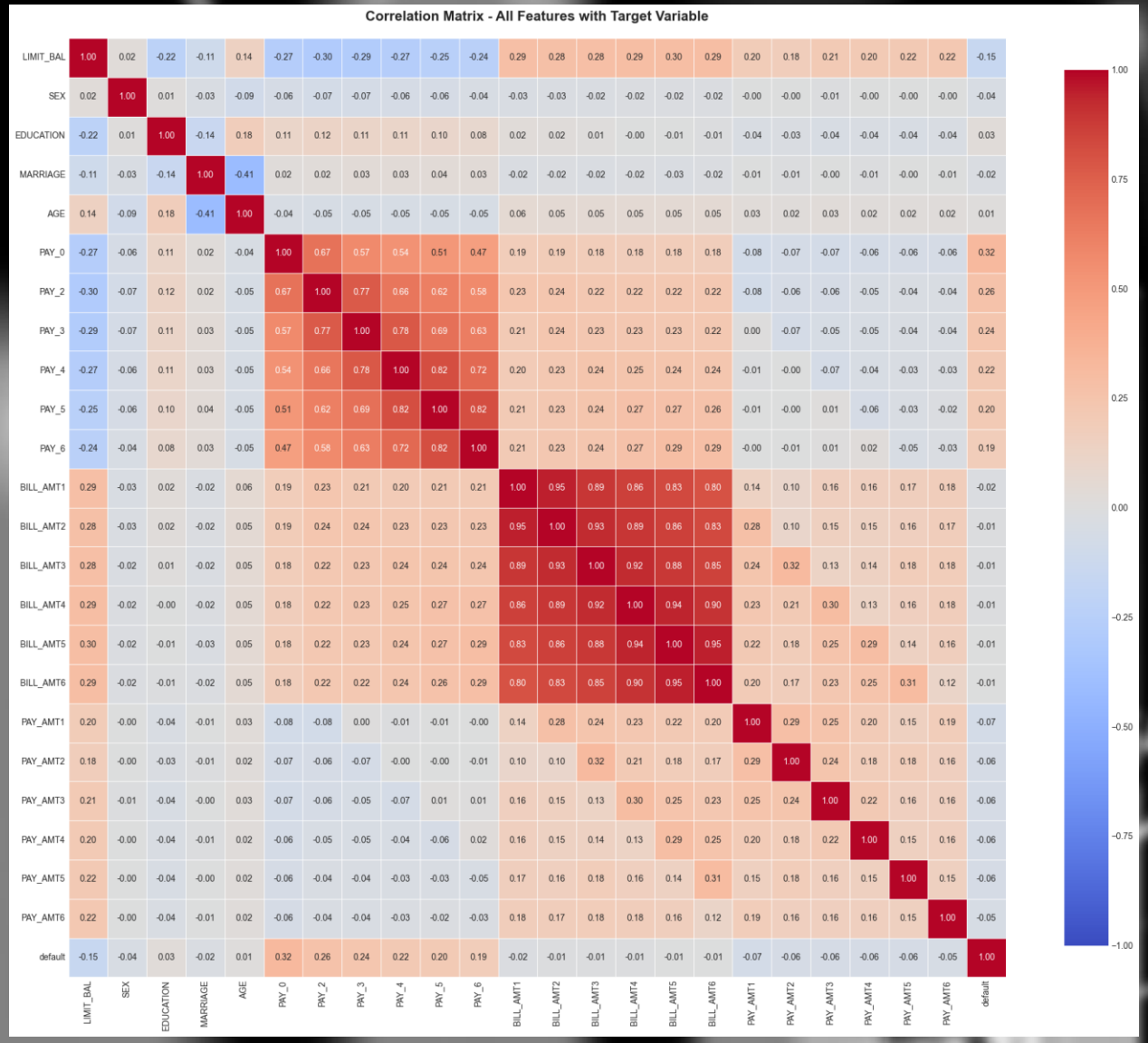
Payment Status

Ex. PAY_AMT1 to PAY_AMT6



Exploratory Data Analysis (EDA)

Correlation Analysis

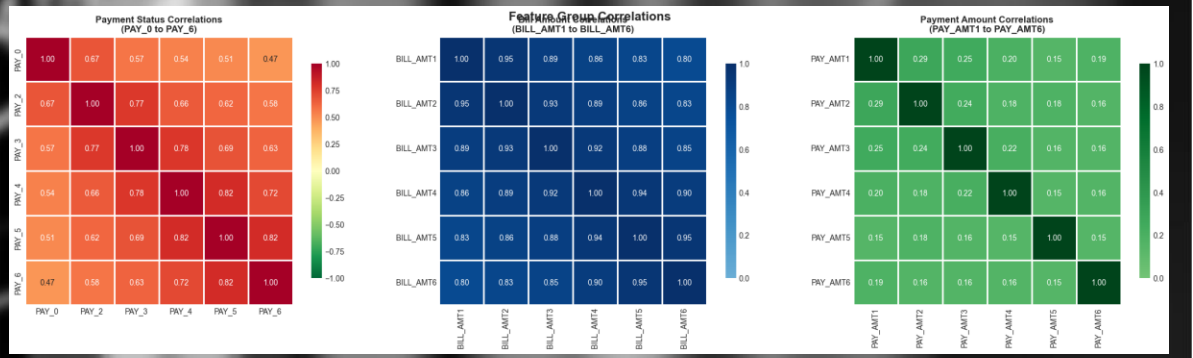


Lessons Learned

We see high correlations for pay features, and demographics.

Once the grand correlation analysis we selected the highest correlations to compare the correlations.

See below.



Exploratory Data Analysis (EDA)

Key Findings

- The dataset is clean and ready for modeling .
- We do need to keep in mind we need to use techniques like class weightings and focus on Recall and F-1 score, not just accuracy
- The most important predictors are payment status and credit utilization, not demographics.
- The class imbalance and temporal nature of the data require special handling in modeling and feature engineering.

SUMMARY

FEATURE ENGINEERING

Feature Engineering

Our Focus & Features

Our Focus

Based on our EDA insights, we engineered features focusing on:

- Credit utilization patterns (strong predictor of default)
- Payment behavior consistency (PAY features = top predictors)
- Temporal trends (bill/payment trajectories)
- Risk indicators (late payments, underpayment patterns)

High Level Features

1. Credit Utilization Features (3 features, i.e. average utilization across 6 months)
2. Payment Fraction Features (3 features)
3. Payment Behavior Features (2 features)
4. Temporal Trend Features (3 features)
5. Repayment Status Features (3 features)
6. Bill Stability Features (2 features)
7. Payment Consistency Features (2 features)
8. Risk Indicator Features (3 features)
9. Demographic Interaction Features (4 features, ex. Education-repayment)

We created 23 new features and verified they aren't missing values or infinite values.

EDA Engineered Features

Total engineered features: 25

	count	mean	std	min	25%	50%	75%	max
util_mean	30000.0000	0.3730	0.3519	-0.2326	0.0300	0.2848	0.6879	5.3640
util_max	30000.0000	0.4950	0.4330	-0.1000	0.0706	0.4307	0.9232	10.6884
util_recent	30000.0000	0.4238	0.4115	-0.6199	0.0220	0.3140	0.8298	6.4550
payfrac_mean	30000.0000	185.5246	1664.3268	0.0000	0.0416	0.0986	1.5294	83050.6667
payfrac_recent	30000.0000	100.2243	2699.3173	0.0000	0.0352	0.0574	0.2942	298887.0000
payfrac_min	30000.0000	0.0461	0.1283	0.0000	0.0000	0.0009	0.0348	1.0276
underpay_count	30000.0000	3.4947	2.2027	0.0000	1.0000	4.0000	6.0000	6.0000
overpay_count	30000.0000	0.7960	1.1686	0.0000	0.0000	0.0000	1.0000	6.0000
bill_trend	30000.0000	4210.1761	33444.8078	-1663964.0000	-2348.0000	0.0000	5115.5000	504644.0000
bill_oldtrend	30000.0000	8141.3944	36159.8611	-450018.0000	-1829.0000	220.0000	11171.2500	1286872.0000
pay_trend	30000.0000	437.8990	20910.6391	-845256.0000	-436.0000	176.5000	1672.2500	503930.0000
repay_max	30000.0000	0.4387	1.3452	-2.0000	0.0000	0.0000	2.0000	8.0000
repay_mean	30000.0000	-0.1824	0.9822	-2.0000	-0.8333	0.0000	0.0000	6.0000
months_late	30000.0000	0.8342	1.5543	0.0000	0.0000	0.0000	1.0000	6.0000
bill_std	30000.0000	12077.7875	20302.1491	0.0000	1549.9352	4579.6622	14352.2596	647788.0511
bill_range	30000.0000	28850.2261	47606.9396	0.0000	4011.0000	11593.5000	33151.7500	1682177.0000
zero_payment_count	30000.0000	1.2299	1.7186	0.0000	0.0000	0.0000	2.0000	6.0000
payment_cv	30000.0000	0.9002	0.6533	0.0000	0.3798	0.7883	1.2982	2.4494
high_util_flag	30000.0000	0.2411	0.4277	0.0000	0.0000	0.0000	0.0000	1.0000
deteriorating_flag	30000.0000	0.4670	0.4989	0.0000	0.0000	0.0000	1.0000	1.0000
chronic_late_flag	30000.0000	0.0863	0.2809	0.0000	0.0000	0.0000	0.0000	1.0000
age_util	30000.0000	13.1022	13.3532	-6.9588	1.0488	9.2619	22.1540	171.6493
limit_per_age	30000.0000	4829.5977	3712.5711	161.2903	1818.1818	4000.0000	6896.5517	26785.7143
edu_repay	30000.0000	-0.2416	1.9525	-12.0000	-1.0000	0.0000	0.0000	16.5000
gender_util	30000.0000	0.5864	0.6066	-0.4652	0.0483	0.4009	0.9338	10.7281

Missing values: 0
Infinite values: 0
All engineered features are clean (no missing/infinite values)

1. months_late : +0.3984 (↑ positive)
2. repay_max : +0.3310 (↑ positive)
3. chronic_late_flag : +0.3106 (↑ positive)
4. repay_mean : +0.2820 (↑ positive)
5. edu_repay : +0.2696 (↑ positive)
6. zero_payment_count : +0.1634 (↑ positive)
7. limit_per_age : -0.1599 (↓ negative)
8. overpay_count : -0.1263 (↓ negative)
9. util_mean : +0.1155 (↑ positive)
10. age_util : +0.1085 (↑ positive)
11. high_util_flag : +0.1080 (↑ positive)
12. underpay_count : +0.0984 (↑ positive)
13. gender_util : +0.0958 (↑ positive)
14. deteriorating_flag : -0.0903 (↓ negative)
15. util_recent : +0.0862 (↑ positive)

Feature Engineering

Insights

General Insights

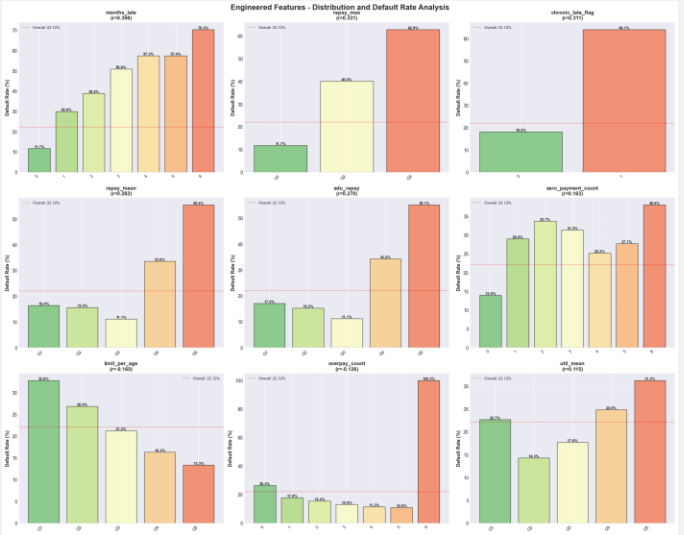
Top 3 engineered features by correlation with default:

1. months_late (count of months with late pay)
2. repay_max (max pay delay)
3. chronic_late_flag (>3 months late)

Observation (not just with top 3):

- Utilization and repayment features show strongest correlations
- Temporal trends capture payment behavior changes
- Risk flags effectively identify high-risk patterns

Data Viz Insights



- Higher values in risk features correlate with higher default rates
- Utilization features show clear risk gradients
- Payment behavior features distinguish defaulters from non-defaulter

Data preparation for model training

```
# Train-Test Split with Stratification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

57] ✓ 0.0s

- TRAIN-TEST SPLIT & PREPROCESSING

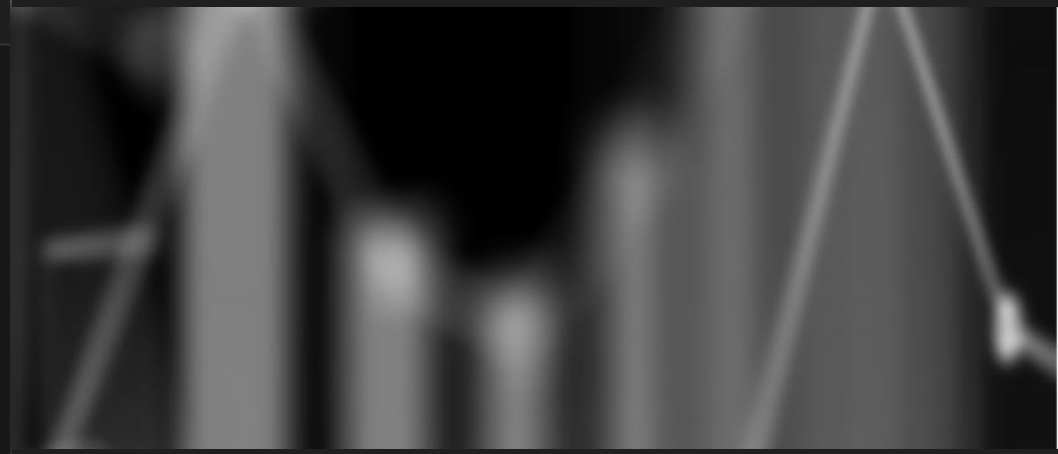
```
# Split with stratification to preserve class distribution (80/20 split per requirement)
X_train, X_test, y_train, y_test = train_test_split(
    X_engineered, y,
    test_size=0.2,
    random_state=RANDOM_STATE,
    stratify=y
)

print(f"\nDataset Split:")
print(f"  Training set: {X_train.shape[0]:,} samples ({X_train.shape[0]/len(X_engineered)*100:.1f}%)")
print(f"  Test set:     {X_test.shape[0]:,} samples ({X_test.shape[0]/len(X_engineered)*100:.1f}%)")
print(f"  Features:     {X_train.shape[1]}")
```

58] ✓ 0.0s

```
Dataset Split:
  Training set: 24,000 samples (80.0%)
  Test set:     6,000 samples (20.0%)
  Features:     48
```

```
Class Distribution:
  Training default rate: 22.12%
  Test default rate:     22.12%
  Stratification preserved class balance
```



```
Features scaled (mean=0, std=1)

Sample scaled values (first 5 features, first sample):
Before: [1.6e+05 2.0e+00 2.0e+00 2.0e+00 3.3e+01]
After:  [-0.05686623  0.80844039  0.18452304  0.85673912 -0.26455769]
```

MODEL DEVELOPMENT AND EVALUATION

XGBoost Baseline

```
print("\n1. Training Baseline XGBoost...")
xgb_baseline = XGBClassifier(
    n_estimators=100,
    max_depth=6,
    learning_rate=0.1,
    random_state=RANDOM_STATE,
    n_jobs=-1,
    eval_metric='logloss'
)
```

General Insights

Model 1: Standard XGBoost Baseline - optimized for accuracy.

Some stats!
Accuracy: 0.8183
Precision: 0.6591
Recall: 0.3700
F1-Score: 0.4739
AUC (Area Under the ROC Curve): 0.7779
Missed Defaults (FN): 836
False Alarms (FP): 254
Caught Defaults (TP): 491
True Negatives (TN): 4419

What is XGBoost?

XGBoost (aka eXtreme Gradient Boosting) uses gradient boosted decision trees (a learning algorithm that uses gradient descent) to reduce overfitting. XGBoost allows:

- Parallelism – speed up!
- Cache-awareness – speed up!
- Regularization (hyperparameter tuning)

XGBoost Baseline – Hyperparameter Tuning and impact

General Insights

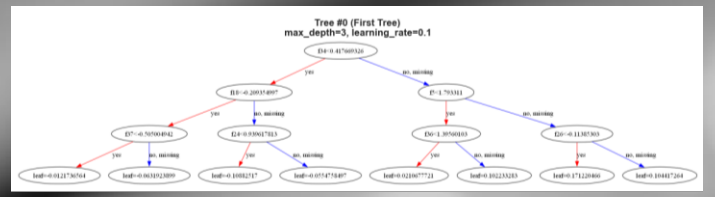
Parameters tuning was done on: max_depth, learning_rate, n_estimators, and scale_pos_weight.

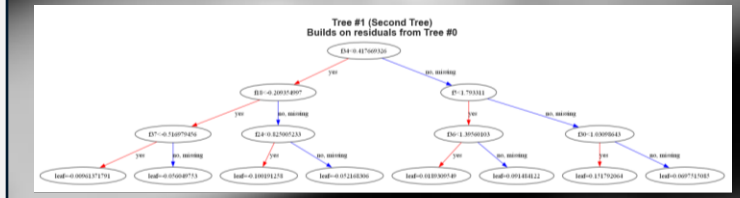
GridSearchCV and XGBClassifier were used together

Accuracy: 0.7747 (from 0.81)
Precision: 0.4923 (from 0.65)
Recall: 0.6051 (from 0.37)
F1-Score: 0.5429 (from 0.47)
AUC: 0.7819 (from 0.77)

Viz

- Max depth per tree: 3
- Each tree is learning from previous tree's errors per the graph
- Leaf values represent contribution to final prediction
- Feature splits chosen to minimize loss function



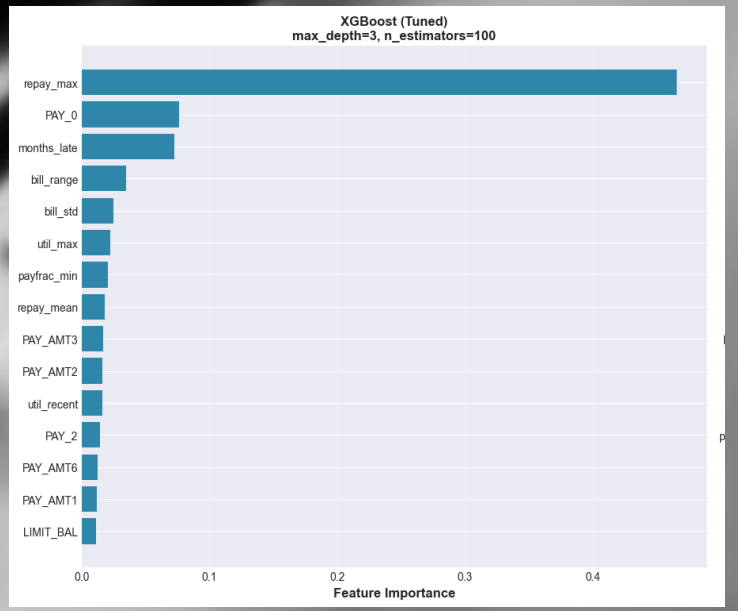


Most Important Features

We can analyze which features the decision trees use most for splitting decisions. For this model it's:

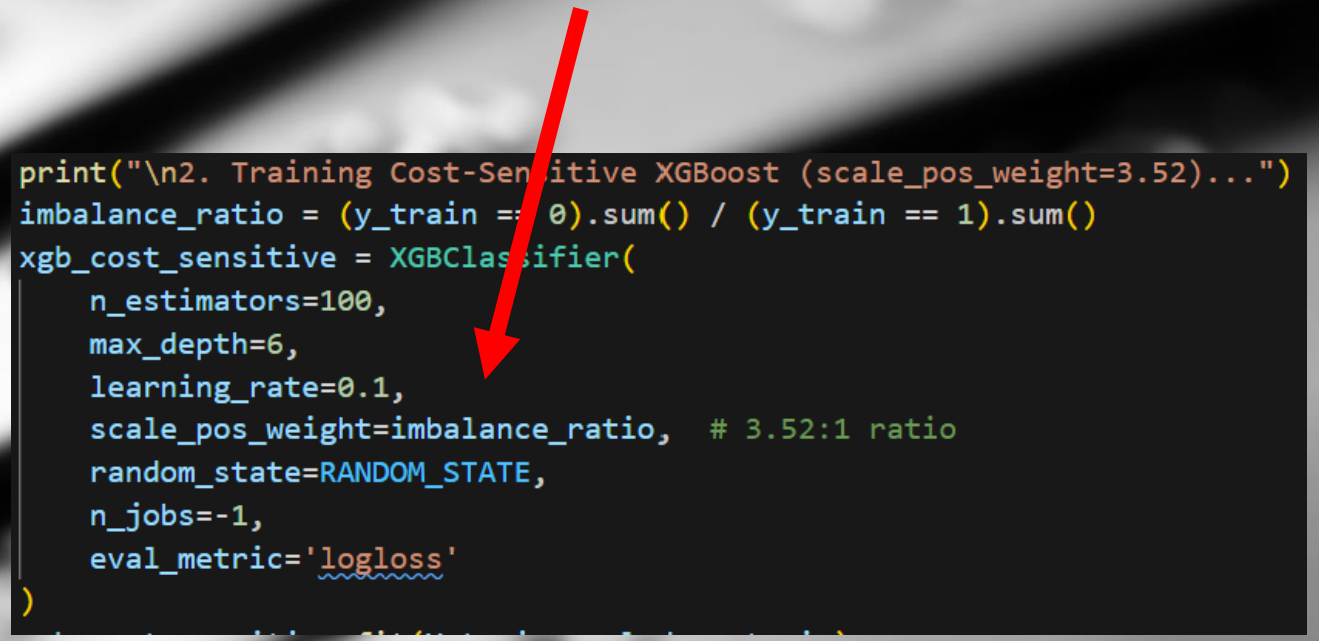
- repay_max, pay, etc.

Feature importance helps interpret what drives default prediction



XGBoost Cost-Effective

Model 2: Cost-effective XGBoost – false negatives are penalized more heavily. To enable cost-sensitive XGBoost, a “scale_pos_weight” was added with a 3.52:1 ratio considering the in-balance of classes.
Accuracy: 0.7620
Precision: 0.4715
Recall: 0.6285
F1-Score: 0.5388
AUC (Area Under the ROC Curve): 0.7757
Missed Defaults (FN): 493
False Alarms (FP): 935
Caught Defaults (TP): 834
True Negatives (TN): 3738



Random Forest Classifier

Model 3: Experiments with other models – Random Forest

Accuracy: 0.7758
Precision: 0.4943
Recall: 0.5923
F1-Score: 0.5389
AUC (Area Under the ROC Curve): 0.7802
Missed Defaults (FN): 541
False Alarms (FP): 804
Caught Defaults (TP): 786
True Negatives (TN): 3869

```
rf_model = RandomForestClassifier(
    n_estimators=100,
    max_depth=10,
    min_samples_split=20,
    min_samples_leaf=10,
    class_weight='balanced',  # Cost-sensitive learning
    random_state=RANDOM_STATE,
    n_jobs=-1
)
rf_model.fit(X_train_scaled, y_train)
rf_time = time.time() - start_time

# Predictions
y_pred_rf = rf_model.predict(X_test_scaled)
y_prob_rf = rf_model.predict_proba(X_test_scaled)[:, 1]

# Evaluation
rf_metrics = {
    'model': 'Random Forest',
    'accuracy': accuracy_score(y_test, y_pred_rf),
    'precision': precision_score(y_test, y_pred_rf),
    'recall': recall_score(y_test, y_pred_rf),
    'f1': f1_score(y_test, y_pred_rf),
    'auc': roc_auc_score(y_test, y_prob_rf),
    'training_time': rf_time,
    'cm': confusion_matrix(y_test, y_pred_rf)
}
```

Random Forest Classifier – Hyperparameter tuning and impact

General Insights

Parameters tuning was done on: n_estimators, max_depth, and min_samples_split

GridSearchCV and RandomForestClassifier were used together

Accuracy: 0.7792 (from 0.7758)
Precision: 0.5096 (from 0.4943)
Recall: 0.5885 (from 0.6923)
F1-Score: 0.5410 (from 0.5389)
AUC: 0.7797(from 0.7802)

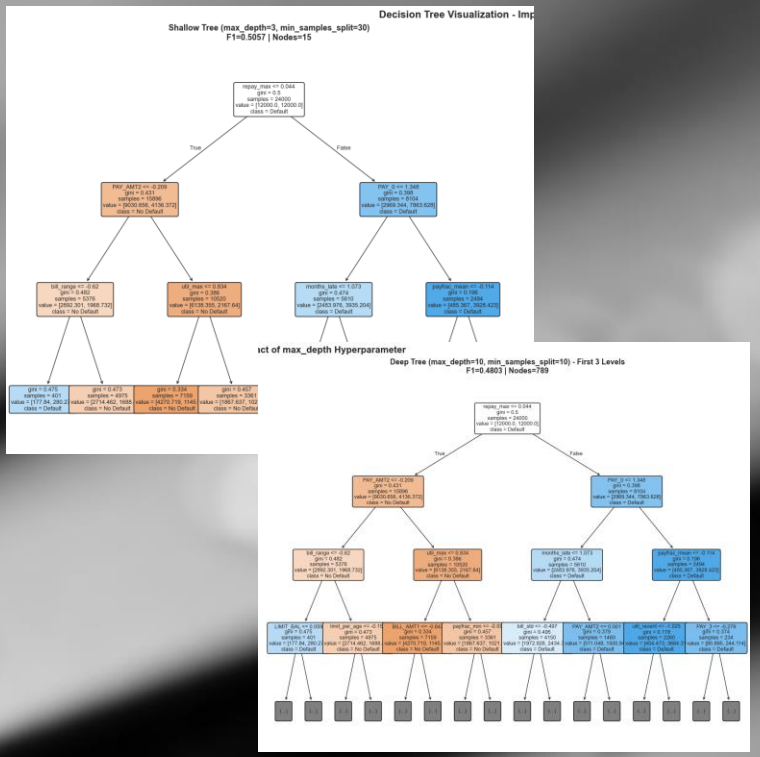
Both models we did tuning on prioritize payment history (PAY_*) and credit utilization features

Viz

Shallow tree (less nodes) are simple but prevents over fitting

Deep tree is more complex but captures patterns

In this case shallow had better F1 score

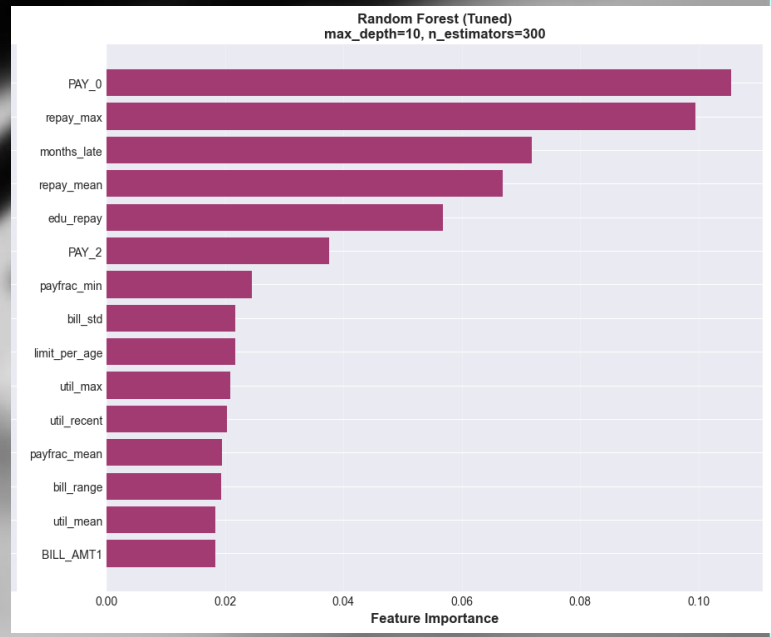


Most Important Features

We can analyze which features the decision trees use most for splitting decisions. For this model it's:

- pay, repay_max etc.

Feature importance helps interpret what drives default prediction



Support Vector Machine

Model 4: More models we tested. This model is also cost sensitive

Accuracy: 0.7528
Precision: 0.4556
Recall: 0.6036
F1-Score: 0.5193
AUC (Area Under the ROC Curve): 0.7576
Missed Defaults (FN): 526
False Alarms (FP): 957
Caught Defaults (TP): 801
True Negatives (TN): 3716

```
svm_model = SVC(
    C=1.0,
    kernel='rbf',
    gamma='scale',
    class_weight='balanced',  # Cost-sensitive learning
    probability=True,  # Enable probability estimates for AUC
    random_state=RANDOM_STATE
)
svm_model.fit(X_train_scaled, y_train)
svm_time = time.time() - start_time

# Predictions
y_pred_svm = svm_model.predict(X_test_scaled)
y_prob_svm = svm_model.predict_proba(X_test_scaled)[:, 1]

# Evaluation
svm_metrics = {
    'model': 'SVM (RBF Kernel)',
    'accuracy': accuracy_score(y_test, y_pred_svm),
    'precision': precision_score(y_test, y_pred_svm),
    'recall': recall_score(y_test, y_pred_svm),
    'f1': f1_score(y_test, y_pred_svm),
    'auc': roc_auc_score(y_test, y_prob_svm),
    'training_time': svm_time,
    'cm': confusion_matrix(y_test, y_pred_svm)
}
```

Kernel Functions Demo (on SVM)

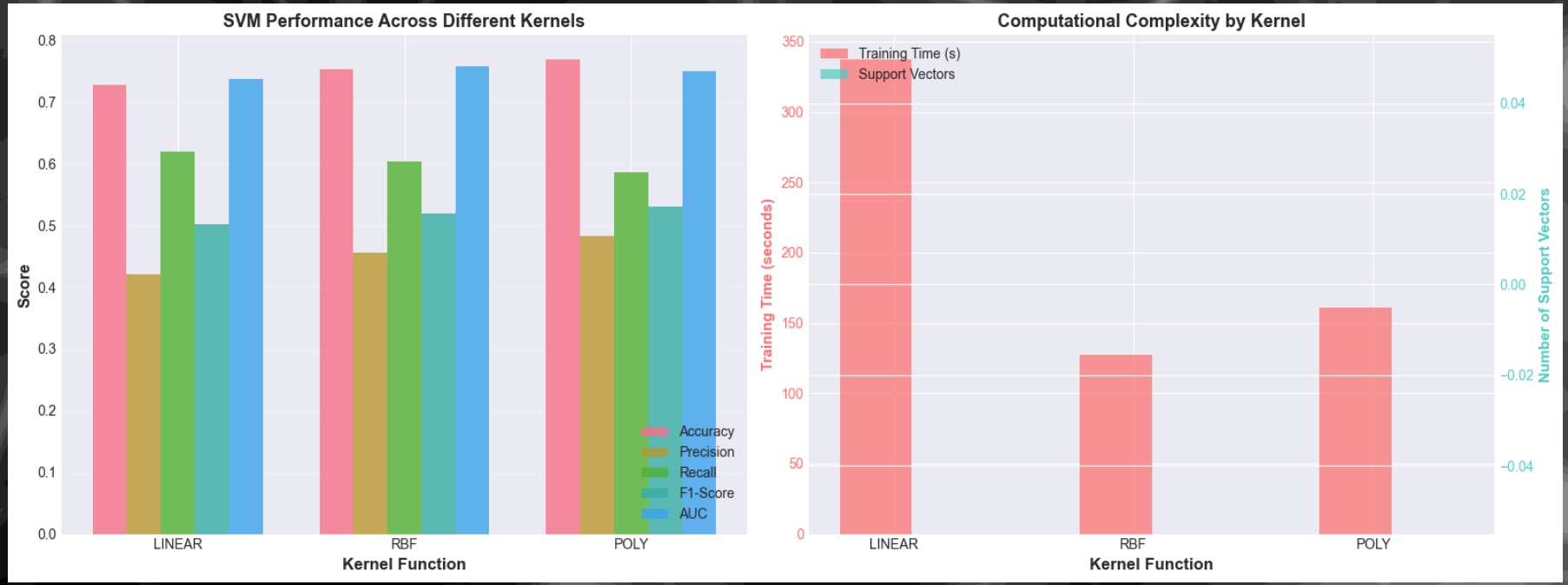
We used Support Vector Machines to show comparing different kernel functions:

- Linear Kernel: For linearly separable data
- Radial Basis Function Kernel: For non-linear patterns
- Polynomial Kernel: For polynomial decision boundaries

Our Insights

We found that

- Linear: Slowest training
- RBF: Most flexible, fastest
- Polynomial: Medium speed, highest f-1 score



Logistic Regression (with SGD)

Model 5: We pefromed logistic regression with a SGD optimizer.

Accuracy: 0.7193
Precision: 0.4126
Recall: 0.6353
F1-Score: 0.5003
AUC (Area Under the ROC Curve): 0.7360
Missed Defaults (FN): 484
False Alarms (FP): 1200
Caught Defaults (TP): 843
True Negatives (TN): 3473

```
start_time = time.time()
sgd_model = SGDClassifier(
    loss='log_loss',  # Logistic regression loss
    penalty='l2',
    alpha=0.0001,
    learning_rate='optimal',
    max_iter=1000,
    class_weight='balanced',  # Cost-sensitive learning
    random_state=RANDOM_STATE,
    n_jobs=-1
)
sgd_model.fit(X_train_scaled, y_train)
sgd_time = time.time() - start_time

# Predictions
y_pred_sgd = sgd_model.predict(X_test_scaled)
# SGD decision_function for probability-like scores
y_prob_sgd = sgd_model.decision_function(X_test_scaled)

# Evaluation
sgd_metrics = {
    'model': 'Logistic Regression (SGD)',
    'accuracy': accuracy_score(y_test, y_pred_sgd),
    'precision': precision_score(y_test, y_pred_sgd),
    'recall': recall_score(y_test, y_pred_sgd),
    'f1': f1_score(y_test, y_pred_sgd),
    'auc': roc_auc_score(y_test, y_prob_sgd),
    'training_time': sgd_time,
    'cm': confusion_matrix(y_test, y_pred_sgd)
}
```

Gradient Boosting – We can skip

Model 6: The out of the box version of Gadient Boosting provided by sklearn

Accuracy: 0.7508
Precision: 0.4546
Recall: 0.6338
F1-Score: 0.5294
AUC (Area Under the ROC Curve): 0.7811
Missed Defaults (FN): 486
False Alarms (FP): 1009
Caught Defaults (TP): 841
True Negatives (TN): 3664

```
start_time = time.time()
gb_model = GradientBoostingClassifier(
    n_estimators=100,
    learning_rate=0.1,
    max_depth=3,
    min_samples_split=20,
    min_samples_leaf=10,
    subsample=0.8,
    random_state=RANDOM_STATE
)

# Gradient Boosting doesn't have class_weight, use sample_weight instead
sample_weights = np.where(y_train == 1, imbalance_ratio, 1.0)
gb_model.fit(X_train_scaled, y_train, sample_weight=sample_weights)
gb_time = time.time() - start_time

# Predictions
y_pred_gb = gb_model.predict(X_test_scaled)
y_prob_gb = gb_model.predict_proba(X_test_scaled)[:, 1]

# Evaluation
gb_metrics = {
    'model': 'Gradient Boosting',
    'accuracy': accuracy_score(y_test, y_pred_gb),
    'precision': precision_score(y_test, y_pred_gb),
    'recall': recall_score(y_test, y_pred_gb),
    'f1': f1_score(y_test, y_pred_gb),
    'auc': roc_auc_score(y_test, y_prob_gb),
    'training_time': gb_time,
    'cm': confusion_matrix(y_test, y_pred_gb)
}
```

Other Analysis

Learning Curves Analysis

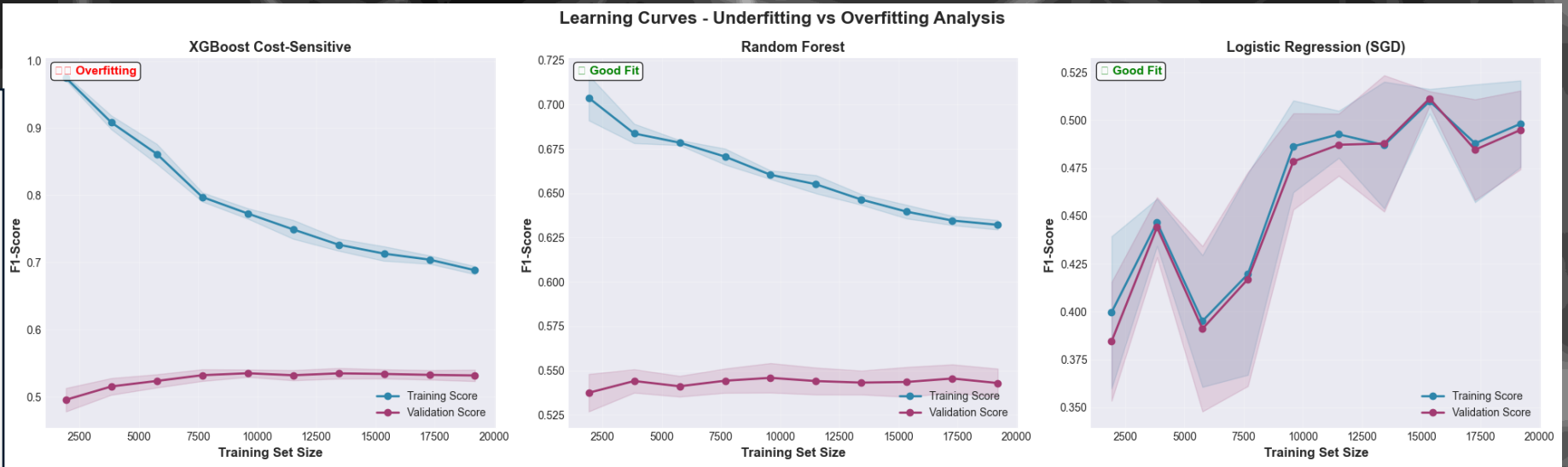
We can use learning curves to detect underfitting and overfitting by showing how well the training and validation scores change with training size.

- Small train-validation gap shows good generalization (low overfitting)
- Converging curves show Model has learned effectively
- Large gap shows overfitting (model memorizes training data)
- Low scores for both show underfitting (model too simple)

Our Insights

Therefore, we can conclude that:

- XGBoost Cost-Sensitive it's overfitting
- Logistic Regression/SGD good fit



K-Fold Cross-Validation

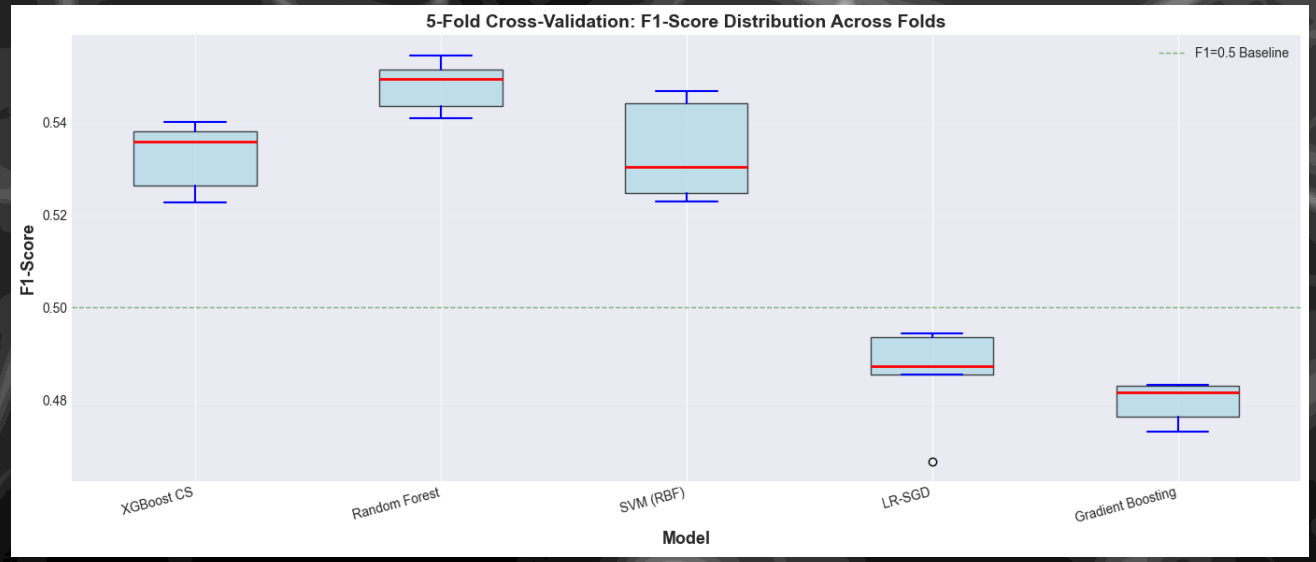
This validation provides performance estimates across different training and validation data splits

- The 22.12% default rate was maintained
- Reduced variance in evaluation
- Low standard deviation shows stable, consistent performance
- High standard deviation shows model sensitive to data splits
- CV scores validate test set results (no lucky splits)

Our Insights

Therefore, we can conclude that:

- SVM is least stable and sensitive to data split
- Random forest sill best
- Logistic Regression seems most stable



Final Findings

Grand Comparison

Model	Accuracy	Precision	Recall	F1-Score	AUC	Missed Defaults (FN)	False Alarms (FP)	Caught Defaults (TP)	True Negatives (TN)
Random Forest	0.7758	0.4943	0.592	0.5389	0.78	541	804	786	3869
XGBoost Cost-Sensitive	0.762	0.4715	0.629	0.5388	0.776	493	935	834	3738
Gradient Boosting	0.7508	0.4546	0.634	0.5294	0.781	486	1009	841	3664
SVM	0.7528	0.4556	0.604	0.5193	0.758	526	957	801	3716
Logistic Regression (SGD)	0.7193	0.4126	0.635	0.5003	0.736	484	1200	843	3473
XGBoost Baseline	0.8183	0.6591	0.37	0.4739	0.778	836	254	491	441

Findings

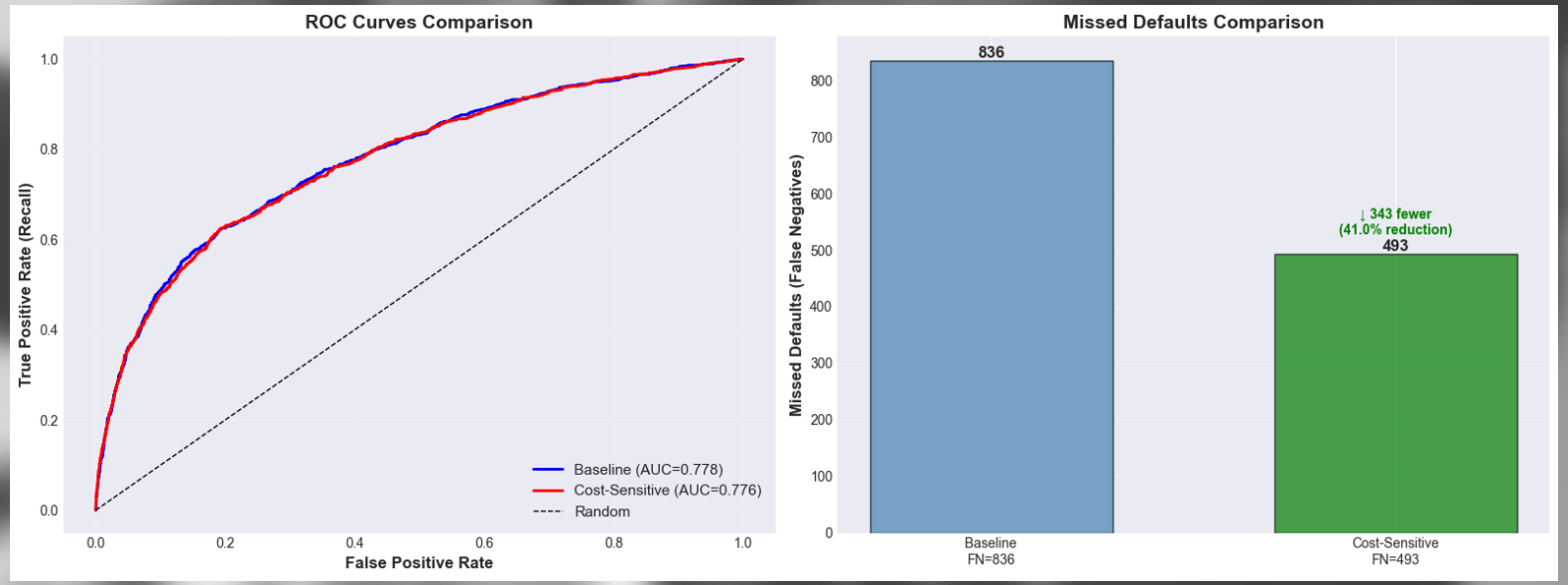
- Cost-sensitive models significantly improve recall (catch more defaults)
- Different models excel at different things
  - XGBoost was most accurate, but had high false negatives
  - Random forest does have higher F-1 score but has for false negatives

Focus Just On XGBoost

Cost Analysis

Baseline Model:
Missed defaults cost:
836 × $10,000 = $8,360,000
False alarm cost:
254 × $200 = $50,800
Total Cost: $8,410,800

Cost-Sensitive Model:
Missed defaults cost:
493 × $10,000 = $4,930,000
False alarm cost:
935 × $200 = $187,000
Total Cost: **$5,117,000**



More Alalyis

From the graphic above we can see both models show similar discrimination ability (close AUCs)

And cost sensitive has much less false positives!

Choice!

Future

Limitations

1. Dataset Constraints
   - 6 month Taiwan-specific data may not generalize globally
   - Features: income, employment status, FICO scores are missing

2. Model Limitations
   - No deep learning methods explored
   - Threshold fixed at 0.5

3. Evaluation Scope
   - Test set performance only (no temporal validation)
   - Binary classification (no risk scoring tiers)
   - Cost assumptions simplified ($10K/default, $200/false alarm)

Future

1. Explore more datasets with:
   - International dataset with more features
   - More demographic features

2. Explore deep learning, ensemble methods (stacking, blending). neural networks (LSTM for temporal patterns). explainable AI (SHAP values for interpretability), time series feature extraction

THANK YOU!