Method for Payment Default Risk Classification

Manvir Singh Cheema, 218033894

March 17, 2025

1 Introduction

The overall goal is to minimize misclassification cost, particularly given the high penalty for misclassifying high-risk orders. Our approach integrates into a cost-sensitive machine learning pipeline:

- Extensive data preprocessing and feature engineering.
- Handling of class imbalance using oversampling techniques. A two-dimensional grid search was performed to compare both SMOTE and ADASYN.
- Model evaluation with a custom cost metric.
- Hyperparameter optimization via a 2D grid search combined with a hill climbing heuristic.
- Analysis of the final predictions.

2 Data Preprocessing and Feature Engineering

The pipeline begins with data integration, cleaning, and transformation. The raw data contains attributes such as customer contact, payment details, order details, and various flags.

2.1 Preprocessing Steps

Training and test datasets are loaded from CSV files. Irrelevant columns (e.g., `ORDER_ID`) are removed from the feature set. Missing values are imputed using an **IterativeImputer** for numeric data and a **SimpleImputer** (using the most frequent value) for categorical data. Date fields (e.g., `DATE_LORDER`) are converted to datetime and decomposed into year, month, and day. Categorical variables are one-hot encoded. Finally, numeric features are scaled using **StandardScaler** (configured without centering due to the sparse format) and reduced in dimensionality using **TruncatedSVD**.

Below is an excerpt from the implementation:

```python
train_order_id = df_train['ORDER_ID']
X_train = df_train.drop(['ORDER_ID', 'CLASS'], axis=1)
y_train = df_train['CLASS']
X_test  = df_test.drop(['ORDER_ID'], axis=1)

if 'DATE_LORDER' in X_train.columns:
    for df in [X_train, X_test]:
        df['DATE_LORDER'] = pd.to_datetime(df['DATE_LORDER'], errors='coerce')
        df['ORDER_YEAR'] = df['DATE_LORDER'].dt.year
        df['ORDER_MONTH'] = df['DATE_LORDER'].dt.month
        df['ORDER_DAY'] = df['DATE_LORDER'].dt.day
        df.drop('DATE_LORDER', axis=1, inplace=True)

num_imputer = IterativeImputer(random_state=42)
X_train[numeric_cols] = num_imputer.fit_transform(X_train[numeric_cols])
X_test[numeric_cols]  = num_imputer.transform(X_test[numeric_cols])
cat_imputer = SimpleImputer(strategy='most_frequent')
```

```python
X_train[cat_cols] = cat_imputer.fit_transform(X_train[cat_cols])
X_test[cat_cols]  = cat_imputer.transform(X_test[cat_cols])
```

Listing 1: Data Preprocessing and Feature Engineering

3 Handling Class Imbalance and Resampling Techniques

Since the dataset is imbalanced with fewer high-risk (`yes`) orders, oversampling techniques are employed. A two-dimensional grid search was performed to explore both SMOTE and ADASYN as potential oversampling methods. During this grid search, using ADASYN resulted in a system crash, while SMOTE produced stable performance and better results on our models. Thus, SMOTE was chosen for the subsequent analysis. The output after applying SMOTE is:

```
CLASS
no     28254
yes    14127
```

```python
resampling_choice = "SMOTE"  # after a 2D search comparing SMOTE and ADASYN
if resampling_choice == "SMOTE":
    from imblearn.over_sampling import SMOTE
    smote = SMOTE(random_state=42, sampling_strategy=0.5)
    X_resampled, y_resampled = smote.fit_resample(X_train_reduced, y_train)
    print("After SMOTE, class distribution:\n", pd.Series(y_resampled).value_counts())
elif resampling_choice == "ADASYN":
    from imblearn.over_sampling import ADASYN
    adasyn = ADASYN(random_state=42, sampling_strategy=0.5)
    X_resampled, y_resampled = adasyn.fit_resample(X_train_reduced, y_train)
    print("After ADASYN, class distribution:\n", pd.Series(y_resampled).value_counts())
elif resampling_choice == "None":
    raise Exception("CHOOSE A RESAMPLING TECHNIQUE")
```

Listing 2: Resampling Technique Selection

4 Cost Function and Model Comparison

A custom cost function is defined to reflect the business requirements: misclassifying a high-risk order as low-risk costs 50, and misclassifying a low-risk order as high-risk costs 5. This function is used to create a scorer for model evaluation.

```python
def cost_metric(y_true, y_pred):
    cm = confusion_matrix(y_true, y_pred, labels=["yes", "no"])
    cost = cm[0,1] * 50 + cm[1,0] * 5
    return cost
cost_scorer = make_scorer(cost_metric, greater_is_better=False)
```

Listing 3: Custom Cost Function Definition

Multiple classifiers are evaluated with cost-sensitive adjustments (using a 10:1 class weight ratio). The classifiers include Decision Tree, KNN (with $k = 1$ and $k = 3$), GaussianNB, Logistic Regression, MLP, and Random Forest.

```python
dense_models = ["DecisionTree", "GaussianNB", "RandomForest", "MLP"]
classifiers = {
    "DecisionTree": DecisionTreeClassifier(random_state=42, max_depth=20, min_samples_leaf
        =50, class_weight={'yes': 10, 'no': 1}),
    "KNN_k1": KNeighborsClassifier(n_neighbors=1),
    "KNN_k3": KNeighborsClassifier(n_neighbors=3),
    "GaussianNB": GaussianNB(),
    "LogisticRegression": LogisticRegression(max_iter=1000, random_state=42, class_weight={'
        yes': 10, 'no': 1}),
    "MLP": MLPClassifier(max_iter=1000, random_state=42),
    "RandomForest": RandomForestClassifier(n_estimators=100, random_state=42, class_weight={
        'yes': 10, 'no': 1})
}
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
results = {}
for name, clf in classifiers.items():
    X_input = X_resampled.toarray() if name in dense_models and hasattr(X_resampled, "
        toarray") else X_resampled
    scores = cross_val_score(clf, X_input, y_resampled, scoring=cost_scorer, cv=cv, n_jobs
        =1)
    avg_cost = -np.mean(scores)
    results[name] = avg_cost
    print(f"{name}:␣average␣cost␣=␣{avg_cost:.2f}")
best_model_name = min(results, key=results.get)
print(f"best␣model␣based␣on␣cost:␣{best_model_name}␣with␣average␣cost:␣{results[
    best_model_name]:.2f}")
```

Listing 4: Classifier Setup

5 Parameter Optimization via a 2D Grid Search and Hill Climbing

A two-dimensional grid search was performed to simultaneously explore different resampling techniques (SMOTE and ADASYN) and model hyperparameters (e.g., varying k in KNN, maximum depth in Decision Trees). During the search, attempts to use ADASYN resulted in a crash, while SMOTE consistently produced more reliable and better results. A hill climbing heuristic was then applied to fine-tune hyperparameters. The best grid search results are summarized in Table 1.

Model	Average Cost	Average Error Rate
Decision Tree	25759.00	0.4056
KNN (k=1)	5721.00	0.1027
KNN (k=3)	8777.00	0.1527
GaussianNB	31923.00	0.6309
Logistic Regression	27997.00	0.6461
MLP	91078.00	0.4108
Random Forest	27715.00	0.0820

Table 1: Grid Search Results for Various Classifiers with SMOTE (ADASYN crashed during execution)

6 Final Model Training and Prediction

Based on both grid search and hill climbing results, the final model is selected. In this case, the best model based on cost is **KNN_k1** with an average cost of 5721.00. The final model is retrained on the entire resampled training data and then used to generate predictions on the test set.

```
if best_model_name in dense_models:
    X_res_final = X_resampled.toarray() if hasattr(X_resampled, "toarray") else X_resampled
else:
    X_res_final = X_resampled
best_model = clone(classifiers[best_model_name])
best_model.fit(X_res_final, y_resampled)
if best_model_name in dense_models:
    X_test_final = X_test_reduced.toarray() if hasattr(X_test_reduced, "toarray") else
        X_test_reduced
else:
    X_test_final = X_test_reduced
test_preds = best_model.predict(X_test_final)
```

Listing 5: Final Model Training and Prediction

7 Outcome Analysis of Predictions

After generating predictions, the results are saved in a file named `prediction.txt`. The following code snippet loads the prediction file and analyzes the outcome by counting the occurrences of each predicted class.

```python
import pandas as pd
file_path = "prediction.txt"
df = pd.read_csv(file_path, delimiter=r'\s+', engine='python')
if 'CLASS' in df.columns:
    yes_count = (df['CLASS'].str.lower() == 'yes').sum()
    no_count = (df['CLASS'].str.lower() == 'no').sum()
    print(f"Count␣of␣yes:␣{yes_count}")
    print(f"Count␣of␣no:␣{no_count}")
else:
    print("Column␣'CLASS'␣not␣found␣in␣the␣file.")
```

Listing 6: Analyzing prediction.txt

The output of this analysis was:

```
Count of yes: 3163
Count of no: 16837
```

8 Conclusion

In this project, the model with the **lowest misclassification cost** was chosen as it directly reflects the cost objectives. A two-dimensional grid search was carried out to compare both SMOTE and ADASYN for handling class imbalance. Due to execution crashes when using ADASYN and consistently better performance with SMOTE across multiple models, SMOTE was selected as the resampling method. This decision was reinforced by the cost matrix, the grid search results, and subsequent fine-tuning via a hill climbing heuristic. The final model (**KNN_k1**) was retrained on the full dataset, and predictions were generated and evaluated, confirming the predicted distribution and overall model performance.