Intelligent Data Analysis and Machine Learning I

Exam: Creditworthiness (Project 4)

Lamia Islam
lamia.islam@uni-potsdam.de
Matrikel Nr. **822947**

Problem Setting

This is a supervised binary classification problem. We have labeled training data where the creditworthiness outcome is known.

Input Space

- Feature vector: $\mathbf{x} \in \mathbb{R}^d$ where $d = 20$ features
- Features include: checking account status, duration, credit history, purpose, credit amount, savings, employment, installment rate, personal status, debtors, residence time, property, age, installment plans, housing, existing credits, job, dependents, telephone, foreign worker status (mixed categorical and numerical features)

Output Space

- Target variable: $y \in \{1, 2\}$ where:
  - $y = 1$: creditworthy
  - $y = 2$: not creditworthy

Dataset Overview

- Dataset: $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{1000}$ where $n = 1000$ samples
- Missing values: Present in features "Purpose", "Present employment since", "Job", and "Foreign worker" (denoted as "?" in the data)

Cost Matrix

$$C = \begin{pmatrix} 0 & 1 \\ 5 & 0 \end{pmatrix}$$

Where:

- $C(1,2) = 1$: Cost of predicting "not creditworthy" when actual is "creditworthy"
- $C(2,1) = 5$: Cost of predicting "creditworthy" when actual is "not creditworthy" (5× more expensive)
- $C(i,i) = 0$: No cost for correct predictions

Objective

Find $f : \mathcal{X} \to \{1,2\}$ that minimizes the expected cost:

$$f^* = \arg\min_f \mathbb{E}_{(\mathbf{x},y)\sim P(\mathbf{X},Y)}[C(y, f(\mathbf{x}))]$$

Importing Necessary Libraries

```
In [1]:
# Core libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from collections import Counter
```

```
In [2]:
# Scikit-Learn libraries
from sklearn.preprocessing import OneHotEncoder
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.utils.class_weight import compute_class_weight
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, StratifiedKFold, GridSearchCV
from sklearn.metrics import (confusion_matrix, ConfusionMatrixDisplay, RocCurveDisplay, roc_curve, auc,
                             precision_recall_curve, average_precision_score)
```

```
In [3]:
# Neural Network libraries
import tensorflow as tf
```

```python
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping
```

In [4]:
```python
# Suppress warnings
import warnings
warnings.filterwarnings('ignore')
import logging
logging.getLogger('tensorflow').setLevel(logging.ERROR)

# Set random seed for reproducibility
tf.random.set_seed(42)
np.random.seed(42)
```

Data Loading

In [5]:
```python
feature_names = [
    'checking_account',   # Status of existing checking account
    'duration',           # Duration in months
    'credit_history',     # Credit history
    'purpose',            # Purpose of credit
    'credit_amount',      # Credit amount
    'savings_account',    # Savings account/bonds
    'employment',         # Present employment since
    'installment_rate',   # Installment rate in percentage of disposable income
    'personal_status',    # Personal status and sex
    'other_debtors',      # Other debtors/guarantors
    'residence_since',    # Present residence since
    'property',           # Property
    'age',                # Age in years
    'other_installments', # Other installment plans
    'housing',            # Housing
    'existing_credits',   # Number of existing credits at this bank
    'job',                # Job
    'dependents',         # Number of people liable to provide maintenance for
    'telephone',          # Telephone
    'foreign_worker',     # Foreign worker
    'target'              # Target variable (1=good, 2=bad)
]

data = pd.read_csv('kredit.dat', sep='\t', header=None, names=feature_names)
data.info()
data.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 21 columns):
 #   Column               Non-Null Count  Dtype 
---  ------               --------------  ----- 
 0   checking_account     1000 non-null   object
 1   duration             1000 non-null   int64 
 2   credit_history       1000 non-null   object
 3   purpose              1000 non-null   object
 4   credit_amount        1000 non-null   int64 
 5   savings_account      1000 non-null   object
 6   employment           1000 non-null   object
 7   installment_rate     1000 non-null   int64 
 8   personal_status      1000 non-null   object
 9   other_debtors        1000 non-null   object
 10  residence_since      1000 non-null   int64 
 11  property             1000 non-null   object
 12  age                  1000 non-null   int64 
 13  other_installments   1000 non-null   object
 14  housing              1000 non-null   object
 15  existing_credits     1000 non-null   int64 
 16  job                  1000 non-null   object
 17  dependents           1000 non-null   int64 
 18  telephone            1000 non-null   object
 19  foreign_worker       1000 non-null   object
 20  target               1000 non-null   int64 
dtypes: int64(8), object(13)
memory usage: 164.2+ KB
```

Out[5]:

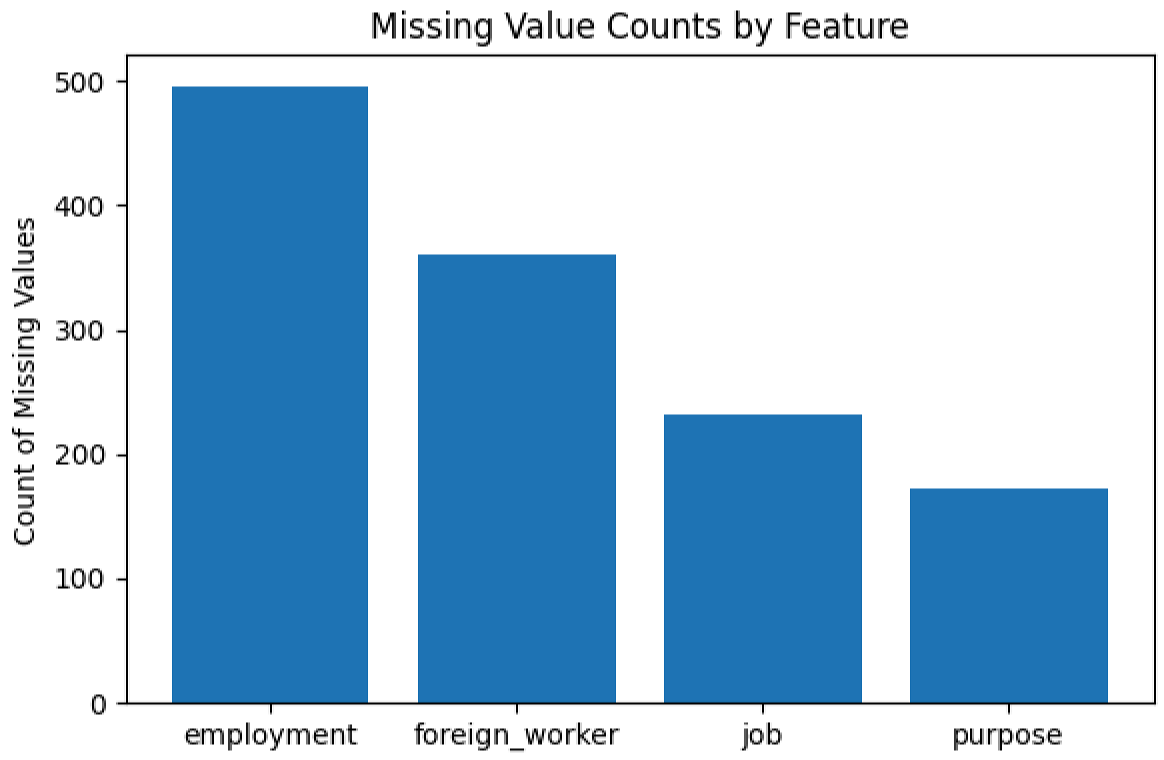
	checking_account	duration	credit_history	purpose	credit_amount	savings_account	employment	installment_rate	personal_status	other_debtors	...	property	age	o
0	A14	36	A32	?	2299	A63	?	4	A93	A101	...	A123	39	
1	A12	18	A32	A46	1239	A65	A73	4	A93	A101	...	A124	61	
2	A13	24	A32	A40	947	A61	A74	4	A93	A101	...	A124	38	
3	A14	15	A33	A43	1478	A61	A73	4	A94	A101	...	A121	33	
4	A14	24	A32	A40	1525	A64	A74	4	A92	A101	...	A123	34	

5 rows × 21 columns

Missing Value Analysis

In [6]:

```
# Identify missing values (marked as '?')
missing_df = pd.DataFrame({
    'Feature': data.columns,
    'Missing_Count': [sum(data[col] == '?') for col in data.columns],
    'Data_Type': data.dtypes
})

missing_df = missing_df.sort_values('Missing_Count', ascending=False)

# Visualize missing values
fig, (ax1) = plt.subplots(1, 1, figsize=(6, 4))

missing_features = missing_df[missing_df['Missing_Count'] > 0]
ax1.bar(missing_features['Feature'], missing_features['Missing_Count'])
ax1.set_title('Missing Value Counts by Feature')
ax1.set_ylabel('Count of Missing Values')
ax1.tick_params(axis='x')

plt.tight_layout()
plt.show()
```

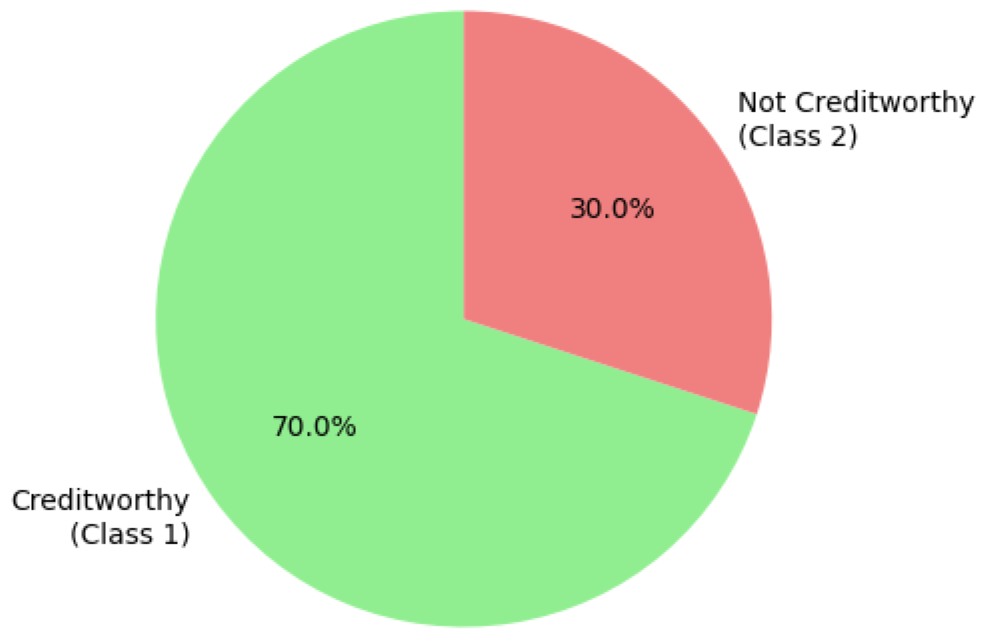


Target Variable Analysis

In [7]:

```
target_counts = data['target'].value_counts()

print("Target Variable Distribution:")
print(f"Creditworthy (1): {target_counts[1]}")
print(f"Not Creditworthy (2): {target_counts[2]}")

# Visualize target distribution
plt.figure(figsize=(5, 5))
labels = ['Creditworthy\n(Class 1)', 'Not Creditworthy\n(Class 2)']
plt.pie(target_counts.values, labels=labels, autopct='%1.1f%%', startangle=90, colors=['lightgreen', 'lightcoral'])
plt.show()
```

```
Target Variable Distribution:
Creditworthy (1): 700
Not Creditworthy (2): 300
```



Feature Distribution Analysis

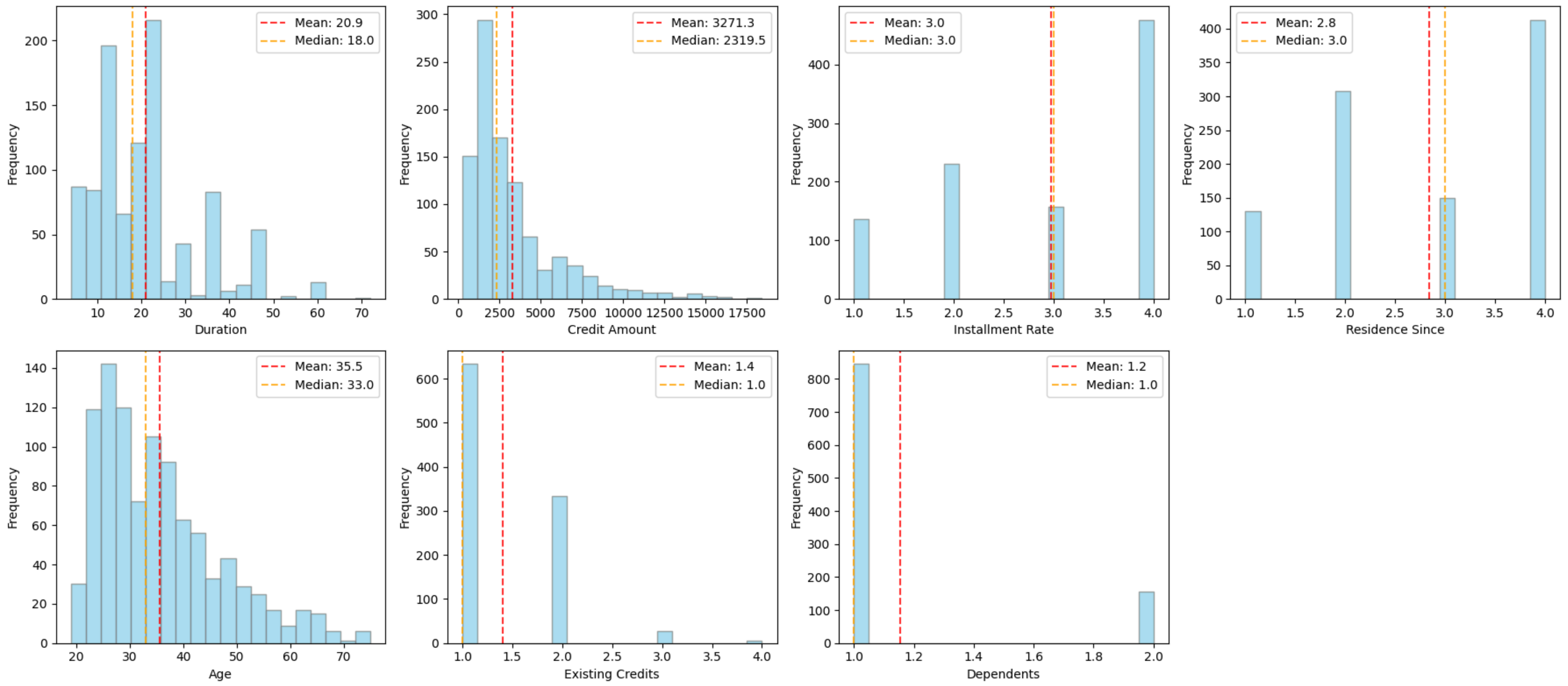
In [8]:
```
# Identify numerical and categorical features
numerical_features = [col for col in data.columns if data[col].dtype == 'int64' and col != 'target']
categorical_features = [col for col in data.columns if data[col].dtype == 'object']

print(f"Numerical features ({len(numerical_features)}): {numerical_features}")
print(f"Categorical features ({len(categorical_features)}): {categorical_features}")
```

```
Numerical features (7): ['duration', 'credit_amount', 'installment_rate', 'residence_since', 'age', 'existing_credits', 'dependents']
Categorical features (13): ['checking_account', 'credit_history', 'purpose', 'savings_account', 'employment', 'personal_status', 'other_debtors', 'property',
'other_installments', 'housing', 'job', 'telephone', 'foreign_worker']
```

Numerical Feature Distribution

In [9]:

```python
fig, axes = plt.subplots(2, 4, figsize=(18, 8))
axes = axes.flatten()

for i, feature in enumerate(numerical_features):
    if i < len(axes):
        ax = axes[i]

        # Get numerical feature data
        feature_data = data[feature]

        # Histogram
        ax.hist(feature_data, bins=20, alpha=0.7, color='Skyblue', edgecolor='gray')
        ax.set_xlabel(feature.replace('_', ' ').title())
        ax.set_ylabel('Frequency')

        # Add statistics
        ax.axvline(feature_data.mean(), color='red', linestyle='--', alpha=0.8, label=f'Mean: {feature_data.mean():.1f}')
        ax.axvline(feature_data.median(), color='orange', linestyle='--', alpha=0.8, label=f'Median: {feature_data.median():.1f}')
        ax.legend()

# Remove empty subplots
for i in range(len(numerical_features), len(axes)):
    fig.delaxes(axes[i])

plt.tight_layout()
plt.show()
```



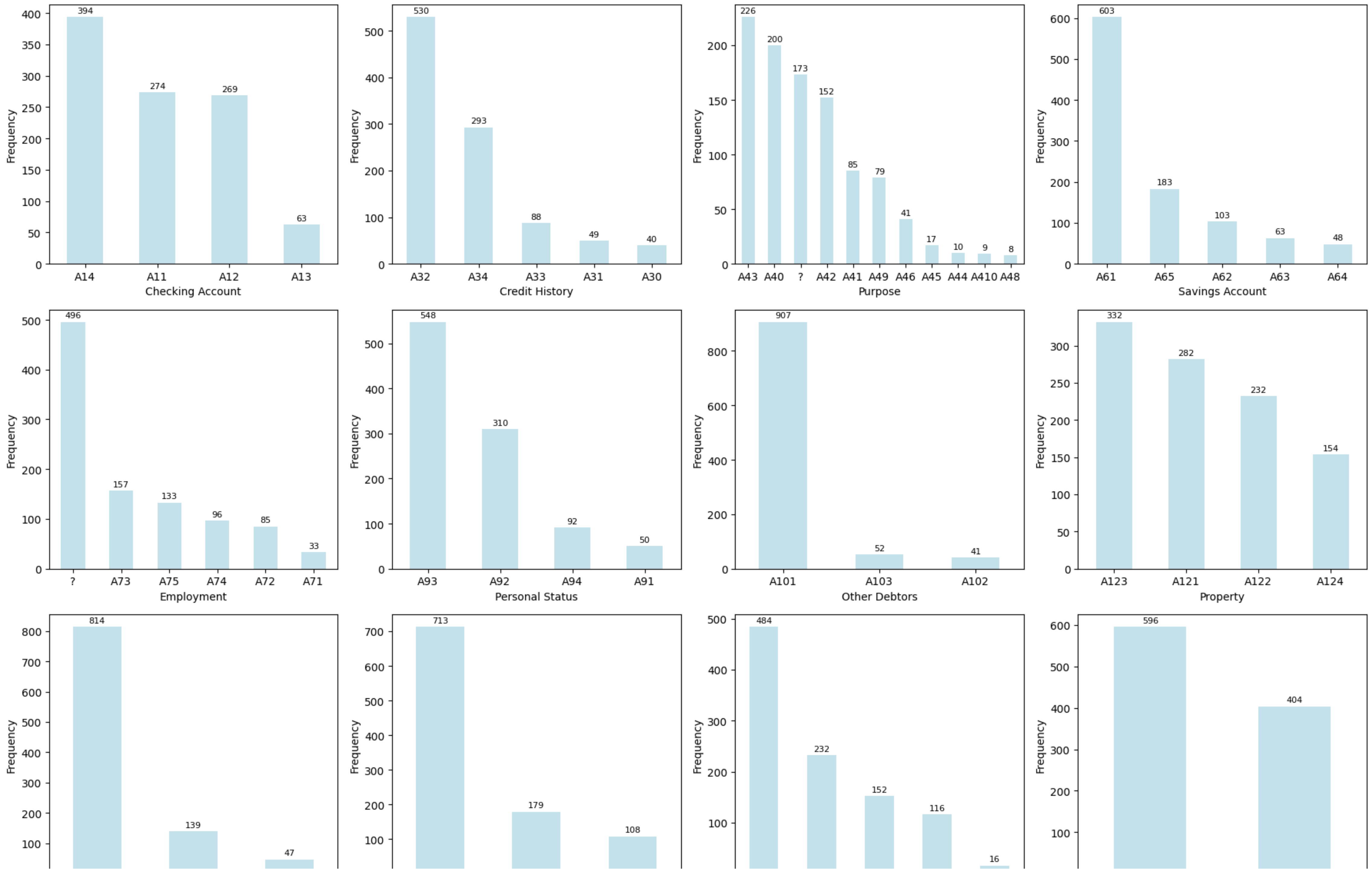
Key Insights

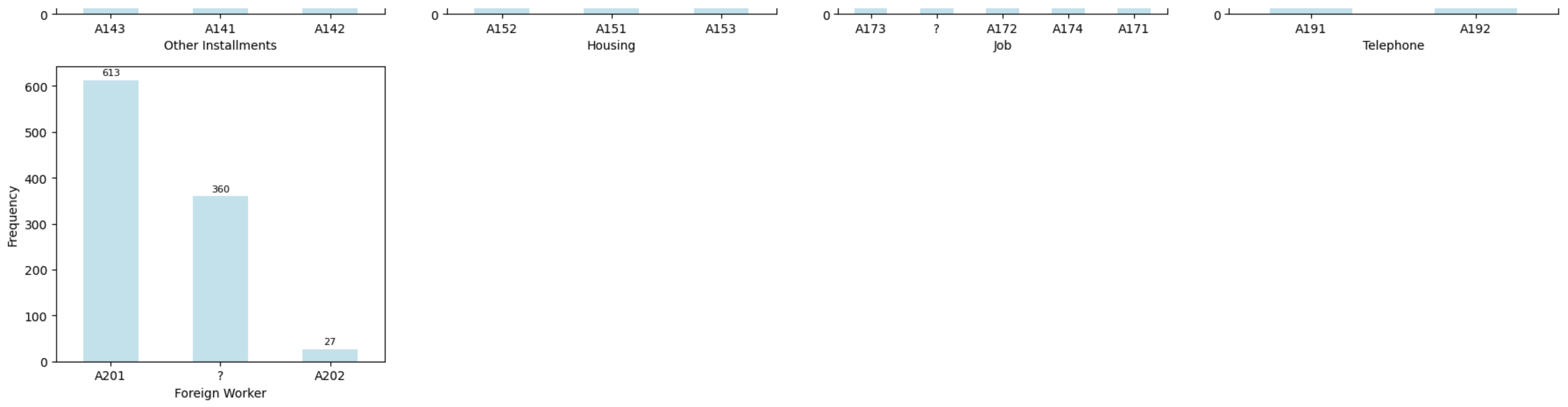
Duration, credit_amount, and age indicate right-skewed distributions. These continuous features require appropriate scaling during preprocessing. Additionally, variables such as installment_rate, residence_since, existing_credits, and dependents show discrete count patterns.

Categorical Feature Distribution

In [10]:
```
n_categorical = len(categorical_features)
n_cols = 4
```

```python
n_rows = (n_categorical + n_cols - 1) // n_cols

fig, axes = plt.subplots(n_rows, n_cols, figsize=(18, 4 * n_rows))
axes = axes.flatten()

for i, feature in enumerate(categorical_features):
    if i < len(axes):
        ax = axes[i]

        # Count values including missing
        feature_counts = data[feature].fillna('Missing').value_counts()

        # Bar plot
        feature_counts.plot(kind='bar', ax=ax, color='lightblue', alpha=0.7)
        ax.set_xlabel(f'{feature.replace("_", " ").title()}')
        ax.set_ylabel('Frequency')
        ax.tick_params(axis='x', rotation=0)

        # Add value counts on bars
        for j, (category, count) in enumerate(feature_counts.items()):
            ax.text(j, count + max(feature_counts) * 0.01, str(count),
                    ha='center', va='bottom', fontsize=8)

# Remove empty subplots
for i in range(len(categorical_features), len(axes)):
    fig.delaxes(axes[i])

plt.tight_layout()
plt.show()
```

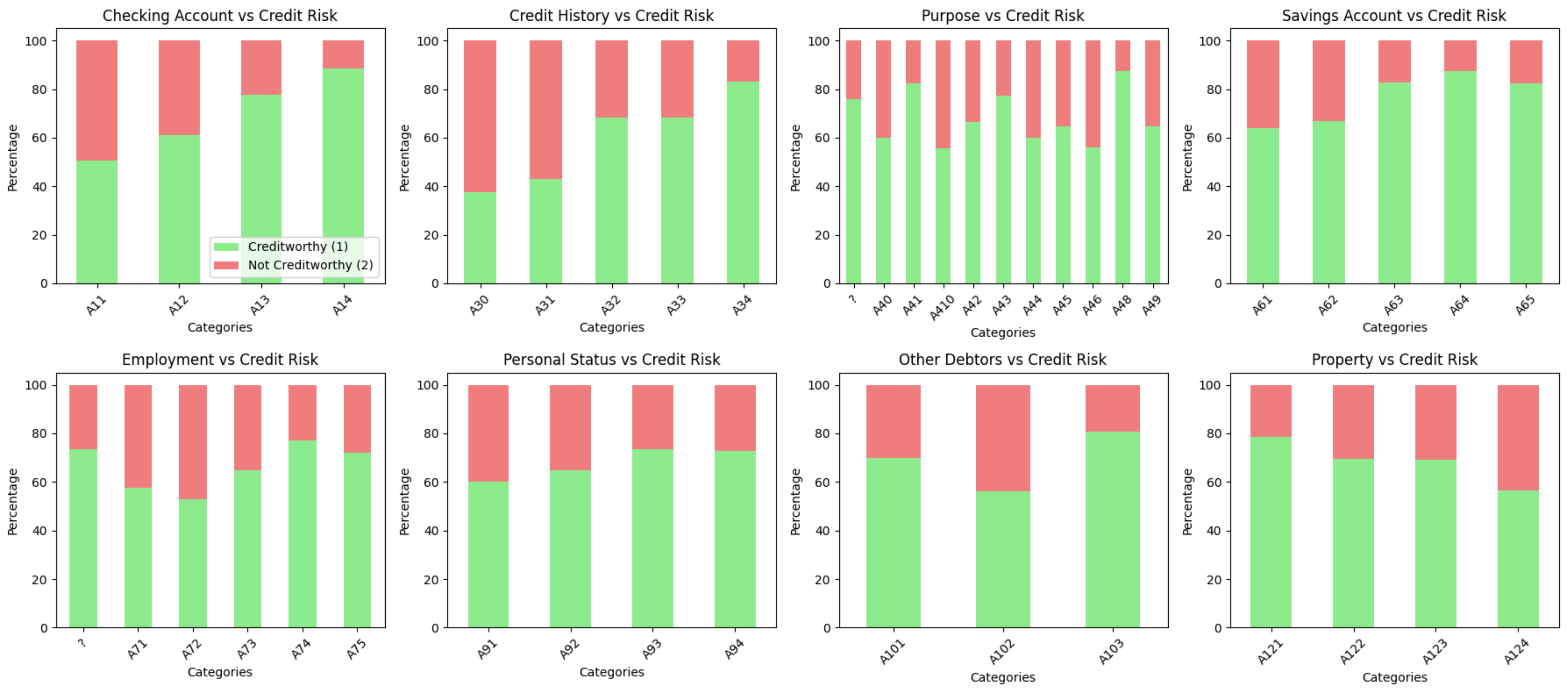

Frequency
Checking Account
A14 394
A11 274
A12 269
A13 63
Credit History
A32 530
A34 293
A33 88
A31 49
A30 40
Purpose
A43 226
A40 200
? 173
A42 152
A41 85
A49 79
A46 41
A45 17
A44 10
A410 9
A48 8
Savings Account
A61 603
A65 183
A62 103
A63 63
A64 48
Employment
? 496
A73 157
A75 133
A74 96
A72 85
A71 33
Personal Status
A93 548
A92 310
A94 92
A91 50
Other Debtors
A101 907
A103 52
A102 41
Property
A123 332
A121 282
A122 232
A124 154
814
139
47
713
179
108
484
232
152
116
16
596
404



Categorical Feature vs Target

```
In [11]:
fig, axes = plt.subplots(2, 4, figsize=(18,8))
axes = axes.flatten()

for i, feature in enumerate(categorical_features):
    if i < len(axes):
        ax = axes[i]

        # Create stacked bar chart
        cross_tab = pd.crosstab(data[feature].fillna('Missing'), data['target'], normalize='index') * 100

        cross_tab.plot(kind='bar', stacked=True, ax=ax,
                       color=['lightgreen', 'lightcoral'],
                       legend=False if i != 0 else True)

        ax.set_title(f'{feature.replace("_", " ").title()} vs Credit Risk')
        ax.set_xlabel('Categories')
        ax.set_ylabel('Percentage')
        ax.tick_params(axis='x', rotation=45)

        if i == 0:
            ax.legend(['Creditworthy (1)', 'Not Creditworthy (2)'], loc='lower right')
```

```
plt.tight_layout()
plt.show()
```



Key Insights

From the visualization, we observe that except Checking Account, other categorical features don't exhibit meaningful ordinal relationships. This insight informs our preprocessing strategy: binary features (telephone and foreign worker) can be encoded using simple 0/1 mapping, while the remaining categorical features will be transformed using one-hot encoding to maintain their non-ordinal nature to avoid any artificial ordering.

Data Preprocessing

We will perform the following preprocessing on the data:

1. For simplicity of calculation, we will convert target values from (1,2) to (1,0), and missing values from '?' to NaN. Then we will create a new dataframe by excluding the target variable to avoid data leakage during model training

2. After that we will first handle the missing values

3. Lastly, we will normalize the numerical features and encode categorical variables

In [12]:
```
# replace 2 as 0
data.target.replace([1,2], [1,0], inplace=True)

# replace '?' with 'NaN'
data.replace("?", np.nan, inplace=True)

label = data['target']
df_new = data.drop(columns=['target'])

print(df_new.shape)
```

```
(1000, 20)
```

Handling Missing Values

In the instruction, it is explicitly mentioned to replace missing values (marked as '?') using linear regression or classification methods. Since all four features with missing values are categorical and three of them are multi-class, we will use **Logistic Regression** for imputation. While other linear classifiers could be used, they would require multiple planes (e.g., one-vs-rest) for multi-class problems, adding more complexity. So that, we will be using a straightforward method that directly addresses our multi-class categorical problem.

In [13]:
```
def impute_missing_values(data, missing_features_df):
  data_imputed = data.copy()
  missing_features_list = missing_features_df["Feature"].tolist()


  for feature in missing_features_list:
    complete_mask = data_imputed[feature].notna()
    incomplete_mask = data_imputed[feature].isna()


    predictor_cols = [col for col in data_imputed.columns if col != feature]
    X_train = data_imputed.loc[complete_mask, predictor_cols]
    y_train = data_imputed.loc[complete_mask, feature]
    X_predict = data_imputed.loc[incomplete_mask, predictor_cols]


    # Simple mode imputation for predictors with missing values
    for col in predictor_cols:
      if X_train[col].isnull().any():
        mode_val = X_train[col].mode().iloc[0]
        X_train[col] = X_train[col].fillna(mode_val)
        X_predict[col] = X_predict[col].fillna(mode_val)


    # One-hot encode categorical predictors
    cat_cols = [c for c in X_train.columns if X_train[c].dtype == "object"]
    encoder = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    encoder.fit(pd.concat([X_train[cat_cols], X_predict[cat_cols]]))

    X_train_enc = encoder.transform(X_train[cat_cols])
    X_predict_enc = encoder.transform(X_predict[cat_cols])

    X_train_final = np.hstack([X_train.drop(columns=cat_cols).to_numpy(), X_train_enc])
    X_predict_final = np.hstack([X_predict.drop(columns=cat_cols).to_numpy(), X_predict_enc])

    # Train model for imputation
    model = LogisticRegression(
      multi_class="multinomial" if len(np.unique(y_train)) > 2 else "auto",
      class_weight="balanced",
      random_state=42,
      max_iter=1000
    )
    model.fit(X_train_final, y_train)
    y_pred = model.predict(X_predict_final)


    data_imputed.loc[incomplete_mask, feature] = y_pred

  return data_imputed
```

```
In [14]: data_imputed = impute_missing_values(df_new, missing_features)
```

```
In [15]: # Check for missing values in the imputed data
         print(f'Number of Missing Values\n{data_imputed.isnull().sum()}')
```

```
Number of Missing Values
checking_account      0
duration              0
credit_history        0
purpose               0
credit_amount         0
savings_account       0
employment            0
installment_rate      0
personal_status       0
other_debtors         0
residence_since       0
property              0
age                   0
other_installments    0
housing               0
existing_credits      0
job                   0
dependents            0
telephone             0
foreign_worker        0
dtype: int64
```

Feature Representation

Numerical Features

We will apply the following transformations to the numerical features:

1. **Log Scaling** for right-skewed continuous features: This transformation reduces skewness, stabilizes variance, and brings the distribution closer to normal.

2. **Min-max Scaling** for all numerical features: This normalization scales the values to a range of [0, 1], ensuring consistency across features.

```
In [16]: def log_scale(data, numeric_cols):
             """
             Apply log scaling to right-skewed numeric features.

             Parameters:
             -----------
             data : pd.DataFrame
                 Input dataframe
             numeric_cols : list of str
                 List of numeric feature names to scale

             Returns:
             --------
             pd.DataFrame
                 Dataframe with scaled numeric features (other columns untouched)
             """
             df_scaled = data.copy()

             # Apply log scaling
             data_scaled = np.log1p(data[numeric_cols])

             # Replace numeric columns with scaled values
             df_scaled[numeric_cols] = pd.DataFrame(data_scaled, columns=numeric_cols, index=data.index)

             return df_scaled
```

```
In [17]: def min_max_scale(data, numeric_cols):
             """
             Apply min-max scaling (0, 1) to numeric features.

             Parameters:
             -----------
             data : pd.DataFrame
                 Input dataframe
             numeric_cols : list of str
                 List of numeric feature names to scale

             Returns:
             --------
             pd.DataFrame
                 Dataframe with min-max scaled numeric features (other columns untouched)
```

```
    """
    df_scaled = data.copy()

    for col in numeric_cols:
        min_val = df_scaled[col].min()
        max_val = df_scaled[col].max()
        df_scaled[col] = (df_scaled[col] - min_val) / (max_val - min_val)

    return df_scaled
```

Categorical Features

Based on the relationship analysis with target variable, most categorical features don't indicate ordinal patterns. Therefore, our encoding strategy is:

1. **Binary encoding** for the binary features: Telephone, Foreign worker
2. **One-hot encoding** for all other categorical features

In [18]:
```
def binary_encoding(data, binary_mappings):
    """
    Apply binary encoding to specified categorical features.

    Parameters:
    -----------
    data : pd.DataFrame
        Input dataframe
    binary_mappings : dict
        Mapping of {column_name: {category_value: binary_value, ...}}
    Returns:
    --------
    pd.DataFrame
        Dataframe with binary encoded features and original columns dropped
    """
    data_binary = data.copy()

    for col, mapping in binary_mappings.items():
        new_col = f"{col}_binary"
        data_binary[new_col] = data_binary[col].map(mapping).astype(int)
        data_binary = data_binary.drop(columns=[col])

    return data_binary
```

In [19]:
```python
def onehot_encoding(data, categorical_cols):
    """
    Apply one-hot encoding to categorical features.

    Parameters:
    -----------
    data : pd.DataFrame
        Input dataframe
    categorical_cols : list of str
        List of categorical feature names to encode

    Returns:
    --------
    pd.DataFrame
        Dataframe containing only the encoded categorical features
    """
    encoder = OneHotEncoder(handle_unknown="ignore", sparse_output=False)

    # Fit + transform categorical features
    data_encoded = encoder.fit_transform(data[categorical_cols])

    # Extract column names
    encoded_cols = encoder.get_feature_names_out(categorical_cols)

    # Return as dataframe
    df_encoded = pd.DataFrame(data_encoded, columns=encoded_cols, index=data.index)

    return df_encoded
```

In [20]:
```python
# Step 1: scale numeric
skewed_cols = ["duration", "credit_amount", "age"]
df_log_scaled = log_scale(data_imputed, skewed_cols)

# Step 2: Min-max scale all numerical features (including log-scaled ones)
df_scaled = min_max_scale(df_log_scaled, numerical_features)

# Step 2: binary encode
binary_mappings = {
    "telephone": {"A191": 0, "A192": 1},
    "foreign_worker": {"A201": 1, "A202": 0}
}
```

```python
df_binary = binary_encoding(df_scaled, binary_mappings)

# Step 3: one-hot encode
categorical_cols = [col for col in df_binary.columns if df_binary[col].dtype == 'object']
df_encoded = onehot_encoding(df_binary, categorical_cols)

# Final combined dataframe
df_final = pd.concat([df_binary.drop(columns=categorical_cols), df_encoded], axis=1)

df_final.head()
```

Out[20]:

	duration	credit_amount	installment_rate	residence_since	age	existing_credits	dependents	telephone_binary	foreign_worker_binary	checking_account_A11	...	ot
0	0.746536	0.515644	1.0	1.000000	0.519211	0.000000	0.0	0	1	0.0	...	
1	0.497945	0.371837	1.0	1.000000	0.847492	0.000000	0.0	0	1	0.0	...	
2	0.600308	0.309334	1.0	0.666667	0.500246	0.000000	1.0	0	1	0.0	...	
3	0.433846	0.412864	1.0	0.666667	0.397474	0.333333	0.0	0	1	0.0	...	
4	0.600308	0.420146	1.0	0.666667	0.419188	0.000000	1.0	1	1	0.0	...	

5 rows × 59 columns

Model Training

Dataset Splitting

For consistency, we have divided the dataset into training and test set before starting training.

In [21]:

```python
X_train, X_test, y_train, y_test = train_test_split(df_final, label, test_size = 0.25, random_state=42)

print(f'X train',X_train.shape)
print(f'X test',X_test.shape)
print(f'y train',y_train.shape)
print(f'y test',y_test.shape)
```

```
X train (750, 59)
X test (250, 59)
y train (750,)
y test (250,)
```

Hyperparameter Tuning

We will use **nested cross-validation** to ensure unbiased model evaluation and prevent data leakage:

- Outer Loop (5-fold): Provides realistic performance estimates by keeping test data completely separate during hyperparameter tuning
- Inner Loop (3-fold): Optimizes hyperparameters using grid search with precision scoring, which directly relates to minimizing false positives (our most costly errors)

Rather than selecting the best parameters from a single fold, we return the most commonly chosen parameters across all folds, because this better reflects the model's general preference for certain hyperparameters across different data subsets.

In [22]:
```python
def nested_cross_validation(clf, param_grid, X, y, outer_splits=5, inner_splits=3):
    """
    Nested cross-validation for hyperparameter tuning to get the best params.

    Parameters
    ----------
    clf : sklearn estimator
        Classifier to evaluate
    param_grid : dict
        Hyperparameter grid for inner CV search
    X : pd.DataFrame
        Feature matrix
    y : pd.Series
        Target labels
    outer_splits : int, default=5
        Number of outer CV folds
    inner_splits : int, default=3
        Number of inner CV folds

    Returns
    -------
    fold_results : list of dict
        Metrics and best params for each outer fold
    common_params : dict
        Most frequently chosen hyperparameters across folds
```

```python
"""

outer_cv = StratifiedKFold(n_splits=outer_splits, shuffle=True, random_state=42)

fold_results = []
best_params_per_fold = []

for fold_idx, (train_idx, test_idx) in enumerate(outer_cv.split(X, y)):
    # Split data
    X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
    y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]

    # Inner CV for hyperparameter tuning
    inner_cv = StratifiedKFold(n_splits=inner_splits, shuffle=True, random_state=42)
    grid = GridSearchCV(clf, param_grid, cv=inner_cv, scoring="precision", n_jobs=-1)
    grid.fit(X_train, y_train)

    best_params = grid.best_params_
    best_params_per_fold.append(best_params)
    best_model = grid.best_estimator_

    # Predictions on test fold
    y_pred = best_model.predict(X_test)
    tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()
    cost = fp * 5 + fn * 1

    # Store fold metrics
    metrics = {
        "fold": fold_idx + 1,
        "cost": cost,
        "best_params": best_params
    }
    fold_results.append(metrics)

    print(f"Fold {metrics['fold']}: Cost={metrics['cost']}, Params={metrics['best_params']}")

# Find most common parameters across folds
common_params = {}
for param in param_grid:
    values = [bp[param] for bp in best_params_per_fold]
    common_params[param] = Counter(values).most_common(1)[0][0]
print("\nMost common parameters across folds:", common_params)

return fold_results, common_params
```

Logistic Regression

We will start with this Linear model, as it sets a simple robust benchmark for comparison. For hyperparameter tuning in Logistic Regression, we will explore:

- **C**: Inverse of regularization strength [0.001, 0.01, 0.1, 1.0, 10.0]
  Helps prevent overfitting by controlling model complexity

- **penalty**: Regularization type ['l1', 'l2']
  L1 can help with feature selection, L2 prevents large coefficients

- **solver**: Algorithm ['liblinear', 'saga']
  Both support L1/L2 regularization and are efficient for our dataset size

- **class_weight**: Class balancing ['balanced']
  Critical for our imbalanced dataset

- **max_iter**: Maximum number of iterations [100, 200, 500]
  Ensures convergence during optimization

In [23]:
```
def hyperParameterTuning_LogisticRegression(features, labels):
    params = {
        "C": [0.001, 0.01, 0.1, 1.0, 10.0],
        "penalty": ["l1", "l2"],
        "solver": ["liblinear", "saga"],
        "class_weight": ["balanced"],
        "max_iter": [100, 200, 500],
        "random_state": [42]
    }

    lr_model = LogisticRegression()
    X, y = features, labels
    fold_results, best_params = nested_cross_validation(lr_model, params, X, y)
    return fold_results, best_params
```

In [24]:
```
lr_fold_results, lr_best_params = hyperParameterTuning_LogisticRegression(X_train, y_train)
```

```
Fold 1: Cost=75, Params={'C': 0.1, 'class_weight': 'balanced', 'max_iter': 100, 'penalty': 'l1', 'random_state': 42, 'solver': 'saga'}
Fold 2: Cost=76, Params={'C': 0.1, 'class_weight': 'balanced', 'max_iter': 100, 'penalty': 'l1', 'random_state': 42, 'solver': 'liblinear'}
Fold 3: Cost=91, Params={'C': 0.1, 'class_weight': 'balanced', 'max_iter': 100, 'penalty': 'l1', 'random_state': 42, 'solver': 'liblinear'}
Fold 4: Cost=114, Params={'C': 0.1, 'class_weight': 'balanced', 'max_iter': 100, 'penalty': 'l1', 'random_state': 42, 'solver': 'saga'}
Fold 5: Cost=101, Params={'C': 0.1, 'class_weight': 'balanced', 'max_iter': 100, 'penalty': 'l1', 'random_state': 42, 'solver': 'liblinear'}

Most common parameters across folds: {'C': 0.1, 'penalty': 'l1', 'solver': 'liblinear', 'class_weight': 'balanced', 'max_iter': 100, 'random_state': 42}
```

In [25]:
```
logreg_model = LogisticRegression(**lr_best_params)
```

Decision Tree

Decision trees are naturally interpretable and handle mixed data types, as well as can capture non linear patterns, making them ideal for understanding credit risk factors. For hyperparameter tuning in Decision Tree, we will explore:

- **max_depth**: Maximum tree depth [2, 5, 10]
  Controls model complexity and prevents overfitting by limiting how deep the tree can grow
- **min_samples_split**: Minimum samples required to split a node [10, 20, 50, 100]
  Prevents overfitting by requiring sufficient data before creating new branches
- **min_samples_leaf**: Minimum samples required in leaf nodes [5, 10, 20, 50]
  Ensures leaf nodes represent meaningful populations and reduces overfitting
- **criterion**: Split quality measure ['gini', 'entropy']
  Gini focuses on class purity, while entropy (information gain) measures information content reduction
- **class_weight**: Class balancing ["balanced"]
  Adjusts node splitting to account for class imbalance

In [26]:
```
def hyperParameterTuning_DecisionTree(features, labels):
    params = {
        "max_depth": [2, 5, 10],
        "min_samples_split": [10, 20, 50, 100],
        "min_samples_leaf": [5, 10, 20, 50],
        "criterion": ["gini", "entropy"],
        "class_weight": ["balanced"],
        "random_state": [42]
    }

    dt_model = DecisionTreeClassifier()
    X, y = features, labels
    fold_results, best_params = nested_cross_validation(dt_model, params, X, y)

    return fold_results, best_params
```

In [27]:
```
dt_fold_results, dt_best_params = hyperParameterTuning_DecisionTree(X_train, y_train)
```

```
Fold 1: Cost=107, Params={'class_weight': 'balanced', 'criterion': 'gini', 'max_depth': 2, 'min_samples_leaf': 50, 'min_samples_split': 10, 'random_state': 4
2}
Fold 2: Cost=72, Params={'class_weight': 'balanced', 'criterion': 'gini', 'max_depth': 2, 'min_samples_leaf': 5, 'min_samples_split': 10, 'random_state': 42}
Fold 3: Cost=90, Params={'class_weight': 'balanced', 'criterion': 'gini', 'max_depth': 2, 'min_samples_leaf': 5, 'min_samples_split': 10, 'random_state': 42}
Fold 4: Cost=111, Params={'class_weight': 'balanced', 'criterion': 'gini', 'max_depth': 5, 'min_samples_leaf': 5, 'min_samples_split': 20, 'random_state': 42}
Fold 5: Cost=102, Params={'class_weight': 'balanced', 'criterion': 'entropy', 'max_depth': 2, 'min_samples_leaf': 5, 'min_samples_split': 10, 'random_state':
42}

Most common parameters across folds: {'max_depth': 2, 'min_samples_split': 10, 'min_samples_leaf': 5, 'criterion': 'gini', 'class_weight': 'balanced', 'random
_state': 42}
```

In [28]:
```
dt_model = DecisionTreeClassifier(**dt_best_params)
```

Random Forest

As we saw Decision Tree performed not so well compared to our baseline model, we want to explore the ensemble version to potentially achieve better performance by combining multiple trees. For hyperparameter tuning in Random Forest, we will explore:

- **n_estimators**: Number of trees in the forest [50, 70, 100, 200]
  More trees generally improve performance but increase computational cost.
- **max_depth**: Maximum tree depth [2, 5, 10, 15]
  Controls model complexity and prevents overfitting by limiting how deep the trees can grow.

- **class_weight**: Class balancing ['balanced', 'balanced_subsample']
  'balanced_subsample' resamples for each tree, while 'balanced' applies globally across the forest.
- **criterion**: Split quality measure ['gini', 'entropy']
  Gini focuses on class purity, while entropy measures information gain.
- **max_features**: Number of features to consider when looking for the best split ['sqrt', 'log2']
  Limits the number of features used in each tree to reduce overfitting.
- **max_samples**: Fraction of samples to train each tree on [0.5, 0.75, 0.8]
  Controls the size of the bootstrap sample for each tree.
- **n_jobs**: Number of jobs to run in parallel [-1]
  Enables parallel processing to speed up training.

In [29]:
```
def hyperParameterTuning_rf(features, labels):
    params = {
        "n_estimators": [50, 70, 100, 200],
        "max_depth": [2, 5, 10, 15],
        "class_weight": ["balanced", "balanced_subsample"],
        "criterion": ["gini", "entropy"],
        "max_features": ["sqrt", "log2"],
        "max_samples": [0.5, 0.75, 0.8],
        "random_state": [42],
        "n_jobs": [-1]
    }

    rf_model = RandomForestClassifier()
    X, y = features, labels
    fold_results, best_params = nested_cross_validation(rf_model, params, X, y)

    return fold_results, best_params
```

In [30]:
```
rf_fold_results, rf_best_params = hyperParameterTuning_rf(X_train, y_train)
```

```
Fold 1: Cost=77, Params={'class_weight': 'balanced', 'criterion': 'gini', 'max_depth': 2, 'max_features': 'sqrt', 'max_samples': 0.5, 'n_estimators': 50, 'n_j
obs': -1, 'random_state': 42}
Fold 2: Cost=82, Params={'class_weight': 'balanced_subsample', 'criterion': 'gini', 'max_depth': 2, 'max_features': 'sqrt', 'max_samples': 0.8, 'n_estimator
s': 200, 'n_jobs': -1, 'random_state': 42}
Fold 3: Cost=110, Params={'class_weight': 'balanced', 'criterion': 'gini', 'max_depth': 2, 'max_features': 'log2', 'max_samples': 0.75, 'n_estimators': 70, 'n
_jobs': -1, 'random_state': 42}
Fold 4: Cost=99, Params={'class_weight': 'balanced', 'criterion': 'gini', 'max_depth': 2, 'max_features': 'sqrt', 'max_samples': 0.75, 'n_estimators': 50, 'n_
jobs': -1, 'random_state': 42}
Fold 5: Cost=98, Params={'class_weight': 'balanced_subsample', 'criterion': 'gini', 'max_depth': 2, 'max_features': 'log2', 'max_samples': 0.75, 'n_estimator
s': 100, 'n_jobs': -1, 'random_state': 42}

Most common parameters across folds: {'n_estimators': 50, 'max_depth': 2, 'class_weight': 'balanced', 'criterion': 'gini', 'max_features': 'sqrt', 'max_sample
s': 0.75, 'random_state': 42, 'n_jobs': -1}
```

In [31]:
```python
rf_model = RandomForestClassifier(**rf_best_params)
```

Feedforward Neural Network

We will also try a neural network to see if it can find hidden patterns in our credit data that simpler models might miss. We will create just two hidden layers to avoid complexity that our small dataset might not support. The workflow will look like:

1. **Input**: Processed features (59) → First hidden layer (e.g. 64 neurons + ReLU)
2. **Pattern Detection**: First layer output → Second hidden layer (32 neurons + ReLU)
3. **Decision**: Second layer output → Output neuron (sigmoid) → Probability
4. **Classification**: If probability > threshold, predict creditworthy

Our neural network needs to work with the same evaluation tools as the other models. So we will create a wrapper that lets us use the same cross-validation, and tune hyperparameters just like with sklearn models.

With only 750 samples, neural networks are actually at a disadvantage compared to simpler models. We're testing whether the ability to capture complex patterns outweighs this limitation in our credit risk problem.

In [45]:
```python
def create_binary_nn_model(input_dim, hidden_units=64, dropout_rate=0.3, learning_rate=0.001):
    """
    Create a binary classification neural network model.

    Parameters:
    - input_dim: Number of input features
    - hidden_units: Number of neurons in first hidden layer
    - dropout_rate: Dropout rate for regularization
    - learning_rate: Learning rate for optimizer
    """
    model = keras.Sequential()

    # First hidden layer
    model.add(Dense(hidden_units, input_dim=input_dim))
    model.add(Activation('relu'))
    model.add(Dropout(dropout_rate))

    # Second hidden layer
    model.add(Dense(hidden_units // 2))
    model.add(Activation('relu'))
    model.add(Dropout(dropout_rate))

    # Output layer for binary classification
    model.add(Dense(1, activation='sigmoid'))

    # Compile model
    model.compile(
        loss='binary_crossentropy',
        optimizer=Adam(learning_rate=learning_rate),
        metrics=['precision']
    )

    return model


class NNWrapper(BaseEstimator, ClassifierMixin):
    def __init__(self, hidden_units=64, dropout_rate=0.3, learning_rate=0.001,
                 epochs=100, batch_size=32, threshold=0.5, patience=10):
        self.hidden_units = hidden_units
        self.dropout_rate = dropout_rate
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.batch_size = batch_size
        self.threshold = threshold
        self.patience = patience

    def fit(self, X, y):
        """
        Fit the neural network model.
        """
        tf.random.set_seed(42)
        np.random.seed(42)

        self.classes_ = np.unique(y)

        #if not hasattr(self, 'model') or self.model is None:
        self.model = create_binary_nn_model(
            X.shape[1], self.hidden_units, self.dropout_rate, self.learning_rate
        )

        # Adjust class weights for class imbalance
        class_weights = compute_class_weight('balanced', classes=self.classes_, y=y)
        class_weight = {0: class_weights[0], 1: class_weights[1]}

        callbacks = [
            keras.callbacks.EarlyStopping(
                monitor='val_loss',
                patience=self.patience,
                restore_best_weights=True,
                verbose=0
            )
        ]

        self.history = self.model.fit(
            X, y,
            epochs=self.epochs,
            batch_size=self.batch_size,
            class_weight=class_weight,
            validation_split=0.2,
            callbacks=callbacks,
            verbose=0
        )

        return self

    def predict(self, X):
        """Make binary predictions using hyperparameter-tuned threshold."""
        probabilities = self.model.predict(X, batch_size=self.batch_size, verbose=0)
        return (probabilities > self.threshold).astype(int).flatten()
    def predict_proba(self, X):
        """Return prediction probabilities for both classes."""
        proba_class_1 = self.model.predict(X, batch_size=self.batch_size, verbose=0).flatten()
        proba_class_0 = 1 - proba_class_1
        return np.column_stack([proba_class_0, proba_class_1])
```

We will explore the following hyperparameters:

- **hidden_units**: Network capacity [32, 64, 128]
  Controls the number of neurons in the first hidden layer, determining how many patterns the network can learn. More neurons increase model complexity but risk overfitting with our small dataset.
- **dropout_rate**: Regularization strength [0.3, 0.4, 0.5]
  Randomly deactivates neurons during training to prevent overfitting. Higher values provide stronger regularization but may reduce learning capacity.
- **learning_rate**: Optimization speed [0.001, 0.01, 0.1]
  Controls how quickly the model updates its weights during training. Lower values ensure stable learning, while higher values speed up convergence but risk overshooting optimal solutions.
- **epochs**: Training duration [30, 50]
  Number of complete passes through the training data. More epochs allow better learning but increase risk of overfitting.
- **batch_size**: Training efficiency [16, 32]
  Number of samples processed before updating model weights. Smaller batches provide more frequent updates and better gradient estimation for our dataset size.

In [46]:
```
def hyperParameterTuning_NeuralNetwork(features, labels):
    params = {
        "hidden_units": [32, 64, 128],
        "dropout_rate": [0.3, 0.4, 0.5],
        "learning_rate": [0.001, 0.01, 0.1],
        "epochs": [30, 50],
        "batch_size": [16, 32]
    }

    nn_model = NNWrapper()
    X, y = features, labels
    fold_results, best_params = nested_cross_validation(nn_model, params, X, y)
    return fold_results, best_params
```

In [47]:
```
# Run hyperparameter tuning
nn_fold_results, nn_best_params = hyperParameterTuning_NeuralNetwork(X_train, y_train)
```

```
Fold 1: Cost=107, Params={'batch_size': 16, 'dropout_rate': 0.3, 'epochs': 50, 'hidden_units': 128, 'learning_rate': 0.01}
Fold 2: Cost=81, Params={'batch_size': 16, 'dropout_rate': 0.4, 'epochs': 50, 'hidden_units': 128, 'learning_rate': 0.01}
Fold 3: Cost=151, Params={'batch_size': 32, 'dropout_rate': 0.3, 'epochs': 50, 'hidden_units': 64, 'learning_rate': 0.1}
Fold 4: Cost=112, Params={'batch_size': 32, 'dropout_rate': 0.4, 'epochs': 30, 'hidden_units': 64, 'learning_rate': 0.001}
Fold 5: Cost=96, Params={'batch_size': 32, 'dropout_rate': 0.5, 'epochs': 30, 'hidden_units': 128, 'learning_rate': 0.01}

Most common parameters across folds: {'hidden_units': 128, 'dropout_rate': 0.3, 'learning_rate': 0.01, 'epochs': 50, 'batch_size': 32}
```

In [48]:
```
nn_model = NNWrapper(**nn_best_params)
```

Model Evaluation

After hyperparameter tuning, we evaluate all models on training and test sets to assess performance and generalization. Our primary goal is minimizing total cost where false positives (predicting creditworthy when actually not creditworthy) cost 5× more than false negatives.

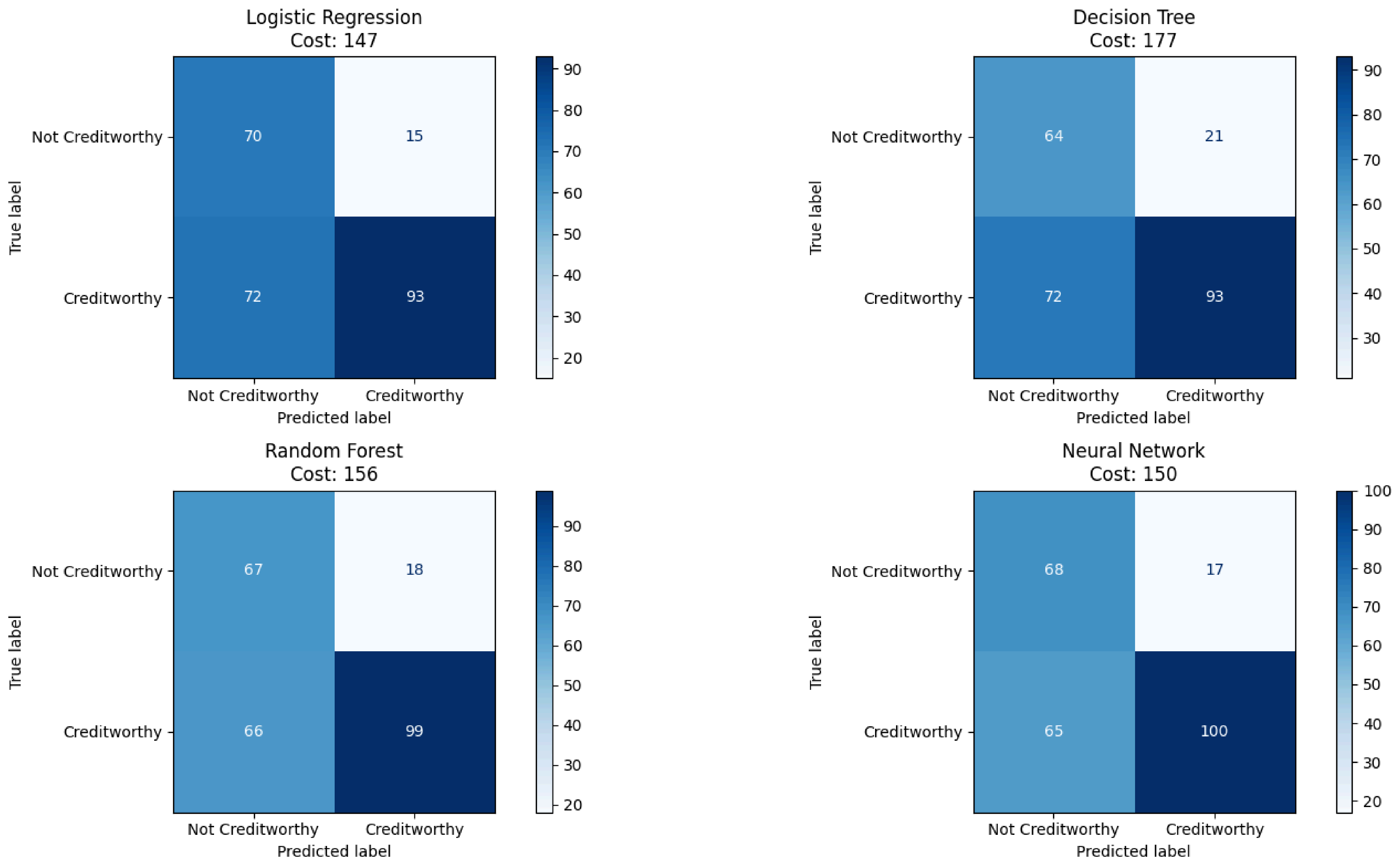
In [36]:
```
models = {
    'Logistic Regression': logreg_model,
    'Decision Tree': dt_model,
    'Random Forest': rf_model,
    'Neural Network': nn_model
}
```

In [37]:
```
def evaluate_all_models(models_dict, X_train, X_test, y_train, y_test):
    """
    Evaluate all models on both training and test sets.

    Parameters:
    -----------
    models_dict : dict
        Dictionary with model names as keys and model objects as values
    X_train, X_test : pd.DataFrame
        Training and test feature sets
    y_train, y_test : pd.Series
```

```
        Training and test target sets
    """

    # Create 2x2 subplots for confusion matrices
    fig, axes = plt.subplots(2, 2, figsize=(15, 8))
    axes = axes.flatten()  # Flatten the 2x2 array for easier indexing

    for idx, (model_name, model) in enumerate(models_dict.items()):
        # Fit model on training data
        model.fit(X_train, y_train)

        # Make predictions
        y_train_pred = model.predict(X_train)
        y_test_pred = model.predict(X_test)

        # Calculate costs
        tn_train, fp_train, fn_train, tp_train = confusion_matrix(y_train, y_train_pred).ravel()
        tn_test, fp_test, fn_test, tp_test = confusion_matrix(y_test, y_test_pred).ravel()

        # Calculate metrics for training set
        train_cost = fp_train * 5 + fn_train * 1
        train_precision = tp_train / (tp_train + fp_train) if (tp_train + fp_train) > 0 else 0
        train_recall = tp_train / (tp_train + fn_train) if (tp_train + fn_train) > 0 else 0
        train_accuracy = (tp_train + tn_train) / len(y_train)

        # Calculate metrics for test set
        test_cost = fp_test * 5 + fn_test * 1
        test_precision = tp_test / (tp_test + fp_test) if (tp_test + fp_test) > 0 else 0
        test_recall = tp_test / (tp_test + fn_test) if (tp_test + fn_test) > 0 else 0
        test_accuracy = (tp_test + tn_test) / len(y_test)

        # Print metrics in table format
        print(f"\n{model_name.upper()}")
        print("-" * 50)
        print("        Cost     Precision   Recall    Accuracy")
        print(f"Train   {train_cost:3d}      {train_precision:.3f}      {train_recall:.3f}     {train_accuracy:.3f}")
        print(f"Test    {test_cost:3d}      {test_precision:.3f}      {test_recall:.3f}     {test_accuracy:.3f}")

        # Confusion Matrix for test set
        ConfusionMatrixDisplay.from_predictions(
            y_test, y_test_pred,
            display_labels=['Not Creditworthy', 'Creditworthy'],
            cmap='Blues',
            ax=axes[idx]
        )
        axes[idx].set_title(f'{model_name}\nCost: {test_cost}')

    plt.tight_layout()
    plt.show()
```

In [51]:
```
# Run evaluation
evaluation_results = evaluate_all_models(models, X_train, X_test, y_train, y_test)
```

```
LOGISTIC REGRESSION
--------------------------------------------------
        Cost    Precision   Recall   Accuracy
Train   431      0.878      0.643     0.681
Test    147      0.861      0.564     0.652

DECISION TREE
--------------------------------------------------
        Cost    Precision   Recall   Accuracy
Train   434      0.877      0.628     0.672
Test    177      0.816      0.564     0.628

RANDOM FOREST
--------------------------------------------------
        Cost    Precision   Recall   Accuracy
Train   404      0.886      0.684     0.712
Test    156      0.846      0.600     0.664

NEURAL NETWORK
--------------------------------------------------
        Cost    Precision   Recall   Accuracy
Train   402      0.885      0.716     0.731
Test    150      0.855      0.606     0.672
```



Logistic Regression
Cost: 147
Decision Tree
Cost: 177
Random Forest
Cost: 156
Neural Network
Cost: 150
True label
Predicted label
Not Creditworthy
Creditworthy
70
15
72
93
64
21
72
93
67
18
66
99
68
17
65
100

Key Observation

Based on the evaluation results and confusion matrices, we can see following characteristics in each model's performance:

Logistic Regression

- Lowest cost among all models: **147**
- When it predicts creditworthy, it's correct **86.1%** of the time
- Captures **56.4%** of actual creditworthy cases

Decision Tree

- Highest cost among all models: **177**
- Lower rprecision (**81.6%**) and same recall (**56.4%**) as Logistic Regression.

Random Forest

- Lower cost than Decision Tree: **156**
- Good precision (**84.6%**) with better recall than Logistic Regression (**60%**)

Neural Network

- Second lowest cost (**150**) and second highest precision (**85.5%**)
- Slightly higher recall than Random Forest (**60.6%**)

In summary, all models show high precision as they are tuned to avoid costly false positive mistakes.

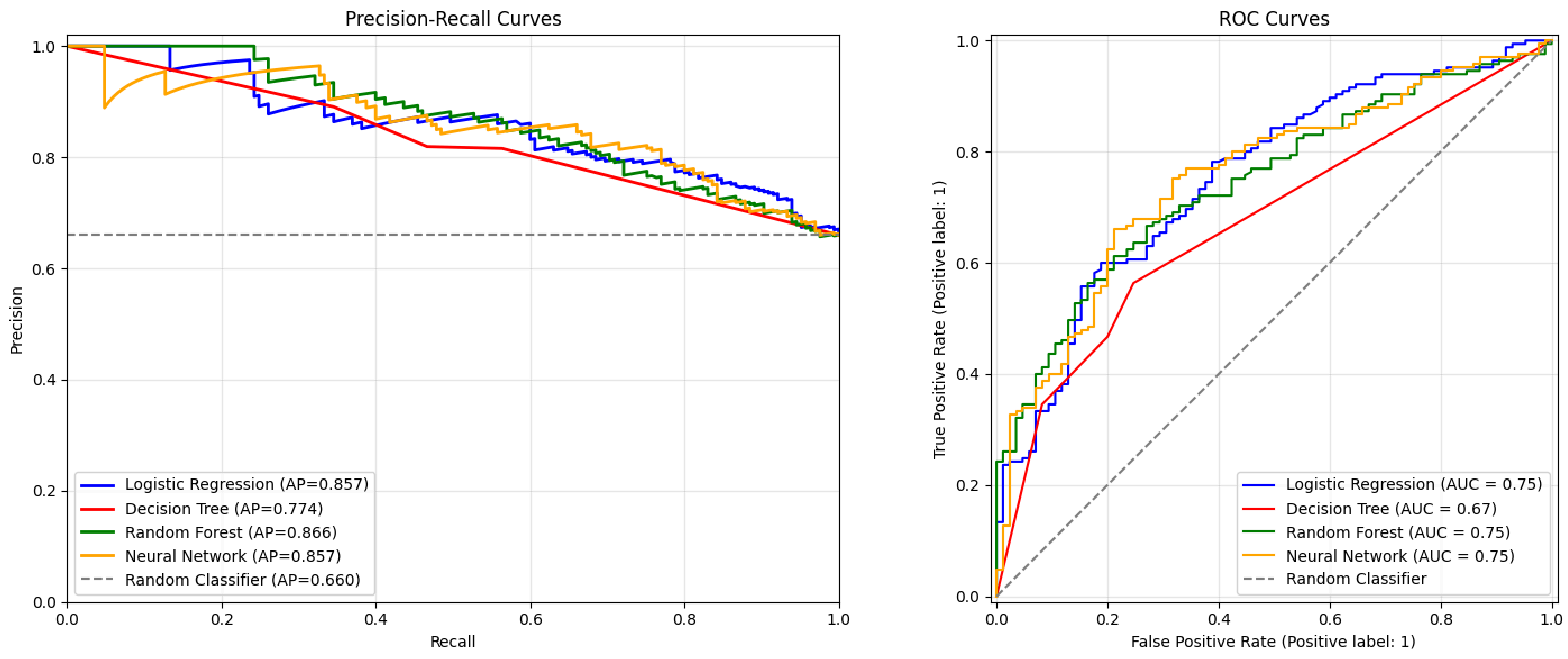
Precision-Recall and ROC-AUC Curves

In [39]:
```
def plot_model_comparison_curves(models_dict, X_test, y_test):
    """
    Plot Precision-Recall and ROC-AUC curves for all models.
    """

    # Set up the plots
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))
```

```python
# Colors for different models
colors = ['blue', 'red', 'green', 'orange', 'purple']

for idx, (model_name, model) in enumerate(models_dict.items()):
    color = colors[idx % len(colors)]

    y_pred_proba = model.predict_proba(X_test)[:, 1]

    # Roc Curve
    roc_display = RocCurveDisplay.from_predictions(
        y_test, y_pred_proba,
        name=f'{model_name}',
        color=color,
        ax=ax2,
    )

    # Precision-Recall curve
    precision, recall, _ = precision_recall_curve(y_test, y_pred_proba)
    avg_precision = average_precision_score(y_test, y_pred_proba)

    ax1.plot(recall, precision, color=color, lw=2,
            label=f'{model_name} (AP={avg_precision:.3f})')


# Precision-Recall plot
ax1.set_xlabel('Recall')
ax1.set_ylabel('Precision')
ax1.set_title('Precision-Recall Curves')
ax1.legend(loc='lower left')
ax1.grid(True, alpha=0.3)
ax1.set_xlim([0.0, 1.0])
ax1.set_ylim([0.0, 1.02])

# Add baseline for PR curve
baseline_precision = sum(y_test) / len(y_test)
ax1.axhline(y=baseline_precision, color='gray', linestyle='--',
            label=f'Random Classifier (AP={baseline_precision:.3f})')
ax1.legend(loc='lower left')

# ROC plot customization
ax2.set_title('ROC Curves')
ax2.grid(True, alpha=0.3)
```

```
    ax2.plot([0, 1], [0, 1], color='gray', linestyle='--', label='Random Classifier')
    ax2.legend(loc='lower right')

    plt.tight_layout()
    plt.show()
```

In [52]:
```
plot_model_comparison_curves(models, X_test, y_test)
```



Key Observation

- Random Forest achieves the highest Average Precision (**86.6%**) and AUC (**75%**).
- Logistic Regression closely follows with an AP of **85.7%** and same AUC of **75%**, offering a simpler and computationally efficient alternative.

- Neural Network shows similar performance as Logistic regression despite having limited data.
- Decision Tree has the lowest AP (**77.4%**) and AUC (**67%**).

All models significantly outperform random classification.

Cross-Validation Evaluation

Performing model evaluation using k-fold cross-validation to ensure robust and unbiased performance metrics across multiple data splits.

In [ ]:
```
def evaluate_all_models_cv(models_dict, X, y, cv_folds=5):
    """
    Evaluate all models using k-fold cross-validation.

    Parameters:
    -----------
    models_dict : dict
        Dictionary with model names as keys and model objects as values
    X : pd.DataFrame
        Feature set
    y : pd.Series
        Target set
    cv_folds : int
        Number of cross-validation folds
    """

    cv = StratifiedKFold(n_splits=cv_folds, shuffle=True, random_state=42)

    # Store results for each model
    results = {}

    for model_name, model in models_dict.items():
        fold_costs = []
        fold_precisions = []
        fold_recalls = []
        fold_accuracies = []

        for fold, (train_idx, test_idx) in enumerate(cv.split(X, y)):
            # Split data for this fold
            X_train_fold = X.iloc[train_idx]
            X_test_fold = X.iloc[test_idx]
            y_train_fold = y.iloc[train_idx]
            y_test_fold = y.iloc[test_idx]

            # Fit and predict
            model.fit(X_train_fold, y_train_fold)
            y_pred_fold = model.predict(X_test_fold)

            # Calculate metrics for this fold
            tn, fp, fn, tp = confusion_matrix(y_test_fold, y_pred_fold).ravel()

            cost = fp * 5 + fn * 1
            precision = tp / (tp + fp) if (tp + fp) > 0 else 0
            recall = tp / (tp + fn) if (tp + fn) > 0 else 0
            accuracy = (tp + tn) / len(y_test_fold)

            fold_costs.append(cost)
            fold_precisions.append(precision)
            fold_recalls.append(recall)
            fold_accuracies.append(accuracy)

        # Calculate mean and std across folds
        results[model_name] = {
            'cost_mean': np.mean(fold_costs),
            'cost_std': np.std(fold_costs),
            'precision_mean': np.mean(fold_precisions),
            'precision_std': np.std(fold_precisions),
            'recall_mean': np.mean(fold_recalls),
            'recall_std': np.std(fold_recalls),
            'accuracy_mean': np.mean(fold_accuracies),
            'accuracy_std': np.std(fold_accuracies)
        }

    # Print results table
    print(f"{'Model':<20} {'Cost':<12} {'Precision':<12} {'Recall':<12} {'Accuracy':<12}")
    print("-" * 80)

    for model_name, metrics in results.items():
        print(f"{model_name:<20} "
              f"{metrics['cost_mean']:.1f}±{metrics['cost_std']:.1f}   "
              f"{metrics['precision_mean']:.3f}±{metrics['precision_std']:.3f}   "
```

```
              f"{metrics['recall_mean']:.3f}±{metrics['recall_std']:.3f}   "
              f"{metrics['accuracy_mean']:.3f}±{metrics['accuracy_std']:.3f}")
```

In [56]:
```
evaluation_cv_results = evaluate_all_models_cv(models, df_final, label, cv_folds=5)
```

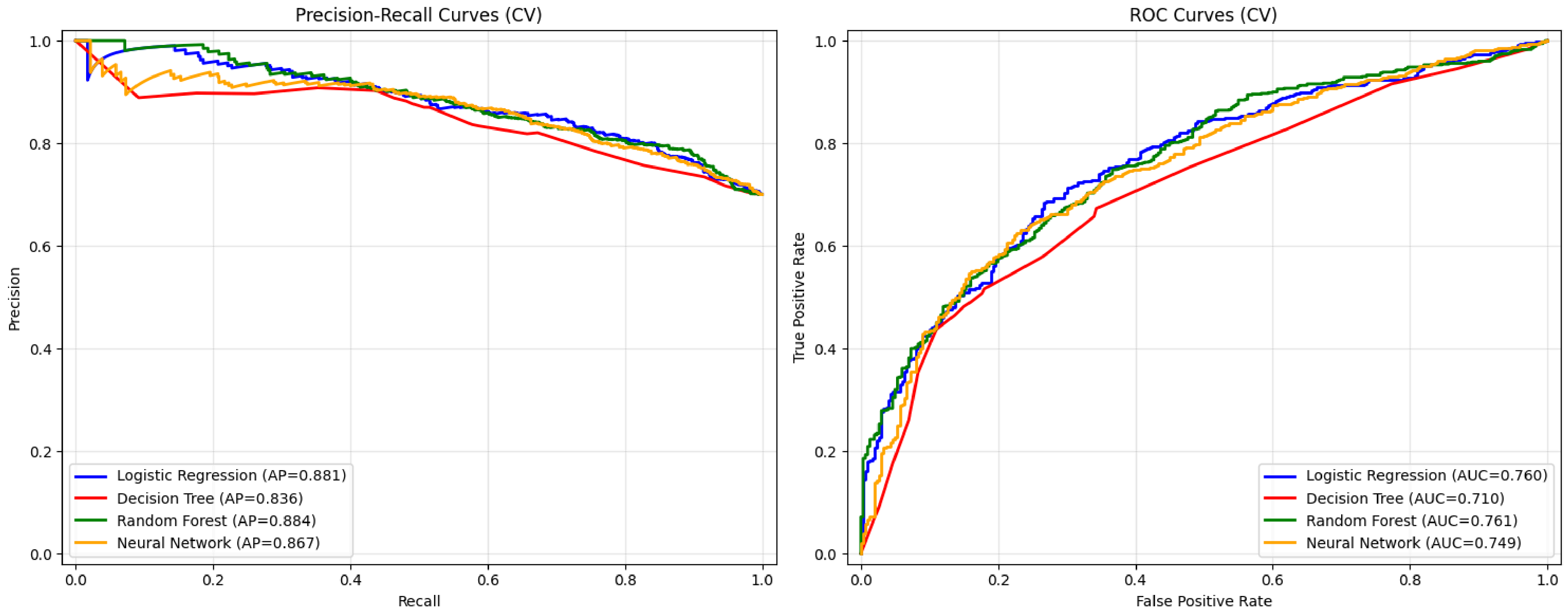
```
Model                    Cost          Precision     Recall        Accuracy
--------------------------------------------------------------------------------
Logistic Regression      123.4±27.1    0.863±0.045   0.654±0.052   0.683±0.029
Decision Tree            122.0±15.9    0.876±0.038   0.507±0.074   0.602±0.032
Random Forest            130.6±23.2    0.851±0.036   0.653±0.034   0.675±0.018
Neural Network           131.6±20.4    0.851±0.030   0.681±0.102   0.690±0.038
```

Key Observation

In the previous single train-test split evaluation, Logistic Regression provided the best overall balance between cost, precision, and recall, while Decision Tree performed the worst. However, the cross-validation (CV) results reveal a different trend. Decision Tree shows improved performance with lower variability across folds, indicating better generalization. Logistic Regression, while still strong, exhibits slightly higher variability in cost across folds. This suggests that Decision Tree may be more robust to data splits than initially observed, while Logistic Regression's performance is more sensitive to the specific train-test split.

In [43]:
```
def plot_model_comparison_curves_cv(models_dict, X, y, cv_folds=5):
    """
    Simple CV curve plotting - shows one averaged curve per model.
    """
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))
    colors = ['blue', 'red', 'green', 'orange', 'purple']
    cv = StratifiedKFold(n_splits=cv_folds, shuffle=True, random_state=42)

    for idx, (model_name, model) in enumerate(models_dict.items()):
        color = colors[idx % len(colors)]

        # Collect all predictions across CV folds
        all_y_true = []
        all_y_proba = []

        for train_idx, test_idx in cv.split(X, y):
            X_train_fold = X.iloc[train_idx]
            X_test_fold = X.iloc[test_idx]
            y_train_fold = y.iloc[train_idx]
            y_test_fold = y.iloc[test_idx]
            # Train and predict
            model.fit(X_train_fold, y_train_fold)
            y_proba_fold = model.predict_proba(X_test_fold)[:, 1]

            # Store results
            all_y_true.extend(y_test_fold)
            all_y_proba.extend(y_proba_fold)

        # Convert to arrays
        all_y_true = np.array(all_y_true)
        all_y_proba = np.array(all_y_proba)

        # Calculate single curves from all CV data
        precision, recall, _ = precision_recall_curve(all_y_true, all_y_proba)
        fpr, tpr, _ = roc_curve(all_y_true, all_y_proba)

        ap = average_precision_score(all_y_true, all_y_proba)
        auc_score = auc(fpr, tpr)

        # Plot curves
        ax1.plot(recall, precision, color=color, lw=2,
                label=f'{model_name} (AP={ap:.3f})')
        ax2.plot(fpr, tpr, color=color, lw=2,
                label=f'{model_name} (AUC={auc_score:.3f})')

    # Customize plots
    ax1.set_xlabel('Recall')
    ax1.set_ylabel('Precision')
    ax1.set_title('Precision-Recall Curves (CV)')
    ax1.legend(loc='lower left')
    ax1.grid(True, alpha=0.3)
    ax1.set_xlim([-0.02, 1.02])
    ax1.set_ylim([-0.02, 1.02])

    ax2.set_xlabel('False Positive Rate')
    ax2.set_ylabel('True Positive Rate')
    ax2.set_title('ROC Curves (CV)')
    ax2.legend(loc='lower right')
    ax2.grid(True, alpha=0.3)
    ax2.set_xlim([-0.02, 1.02])
    ax2.set_ylim([-0.02, 1.02])
    plt.tight_layout()
    plt.show()
```

In [54]:
```
plot_model_comparison_curves_cv(models, df_final, label, cv_folds=5)
```



Key Observation

- **Random Forest** further strengthens its lead, achieving the highest Average Precision (**88.4%**) and AUC (**76.1%**)
- **Logistic Regression** remains highly competitive, with AP (**88.1%**) and AUC (**76%**)
- **Decision Tree** improves significantly in CV (**83.6%** AP, **71%** AUC) compared to the single split (**77.4%** AP, **67%** AUC), suggesting better generalization and reduced variability across folds.
- **Neural Network** shows same performance.

Cross-validation reveals that Random Forest and Logistic Regression not only outperform other models but also maintain their superiority across different data splits. Decision Tree benefits from CV, showing improved and more stable results, while Neural Network remains same. All models continue to significantly outperform random classification.

Conclusion

Logistic Regression emerges as the optimal model for this cost-sensitive credit risk assessment, delivering the best overall balance of cost-effectiveness and predictive performance. While Decision Tree achieves the lowest cost (122) in CV, Logistic Regression follows closely (123.4), making the cost difference negligible. It significantly outperforms Decision Tree in both Average Precision and AUC across cross-validation and single split results.

Although Random Forest achieves the highest predictive metrics (88.4% AP, 76.1% AUC), the difference is very low compared to Logistic Regression (88.1% AP, 76% AUC). The minimal performance gap cannot justify Random Forest's higher cost (130.6).

Neural Network, despite the complexity, fails to outperform the simpler models, highlighting the challenges of applying deep learning to limited datasets where simpler models often prove more effective.

For a balance of cost, precision, interpretability, and computational efficiency, **Logistic Regression** provides the winning combination.