LOAN DEFAULTERS CLASSIFICATION

Dataset Description:

Column Name	Description
earliest_cr_line	The month the borrower's earliest reported credit line was opened
emp_title	The job title supplied by the Borrower when applying for the loan.
fico_range_high	The upper boundary range the borrower’s FICO at loan origination belongs to.
fico_range_low	The lower boundary range the borrower’s FICO at loan origination belongs to.
Grade	LC assigned loan grade
application_type	Indicates whether the loan is an individual application or a joint application with two co-borrowers
initial_list_status	The initial listing status of the loan. Possible values are – W, F
num_actv_bc_tl	Number of currently active bankcard accounts.
mort_acc	Number of mortgage accounts.
tot_cur_bal	Total current balance of all accounts
open_acc	The number of open credit lines in the borrower's credit file.
pub_rec	Number of derogatory public records
pub_rec_bankruptcies	Number of public record bankruptcies.
Purpose	A category provided by the borrower for the loan request.
revol_bal	Total credit revolving balance
Title	The loan title provided by the borrower
total_acc	The total number of credit lines currently in the borrower's credit file
verification_status	Indicates if income was verified by LC, not verified, or if the income source was verified
addr_state	The state provided by the borrower in the loan application
annual_inc	The self-reported annual income provided by the borrower during registration.
emp_length	Employment length in years. Possible values are between 0 and 10 where 0 means less than one year and 10 means ten or more years.
home_ownership	The home ownership status provided by the borrower during registration. Our values are: RENT, OWN, MORTGAGE, OTHER.
int_rate	Interest Rate on the loan
loan_amnt	The listed amount of the loan applied for by the borrower. If at some point in time, the credit department reduces the loan amount, then it will be reflected in this value.
sub_grade	LC assigned loan subgrade
Term	The number of payments on the loan. Values are in months and can be either 36 or 60.
revol_util	Revolving line utilization rate, or the amount of credit the borrower is using relative to all available revolving credit.
Target	loan_status Status of the loan

Table of Content

- 1: Import Libraries
- 2: Read Dataset
- 3: Dataset Overview
  - 3.1: Summary Statistics of Numeric Features
  - 3.2: Summary Statistics of Categorical Features

- 4: EDA
  - 4.1:Univariate Analysis
    - 4.1.1 Numerical Features Univariate Analysis
    - 4.1.2 Categorical Features Univariate Analysis
      - 4.1.2.1: Low Cardinality Categorical Features
      - 4.1.2.2: High Cardinality Categorical Features
  - 4.2: Bivariate Analysis
    - 4.2.1: Numerical Features vs Target (loan_status)
    - 4.2.2: Categorical Features vs Target (loan_status)
      - 4.2.2.1: Low Cardinality Categorical Features vs Target (loan_status)
      - 4.2.2.2: High Cardinality Categorical Features vs Target (loan_status)
- 5: Data Preprocessing
  - 5.1: Feature Selection and Engineering
    - 5.1.1: High Cardinal Categorical Features
    - 5.1.2: Low Cardinal Categorical Features
  - 5.2: Handle Missing Values
  - 5.3: Outlier Treatment
  - 5.4: Duplicated Data
  - 5.5: Encode Categorical Variables
  - 5.6: Check unbalance
    - 5.6.1: Handling unbalance
- 6:Busimess Requirement
- 7: Spliting Training Dataset
- 8: Decision Tree Mode Building
  - 8.1: DT Base Model Definition
  - 8.2: DT Hyperparameter Tuning
  - 8.3: DT Model Evaluation
  - 8.4: Target Prediction for Test Data
- 9: Random Forest Mode Building
  - 9.1: RF Base Model Definition
  - 9.2: RF Hyperparameter Tuning
  - 9.3: RF Model Evaluation
  - 9.4: Target Prediction for Test Data
- 10: XGBoost Mode Building
  - 10.1: XGBoost Base Model Definition
  - 10.2: Outlier treatment using Box-cox
  - 10.3: XGBoost Hyperparameter Tuning
  - 10.4: XGBoost Model Evaluation
  - 10.5: Target Prediction for Test Data
- 11:Conclusion

1: Import Libraries

⬆️ Table Contents

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.filterwarnings('ignore')

from matplotlib.colors import ListedColormap, LinearSegmentedColormap
from sklearn.model_selection import train_test_split, GridSearchCV, StratifiedKFold, c

from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier

from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score, r
from sklearn.metrics import classification_report, RocCurveDisplay, ConfusionMatrixDis

import xgboost as xgb
%matplotlib inline
```

2: Read Dataset

⬆️ Table Contents

In [2]:
```
# Load training and testing datasets
train_df = pd.read_csv("train_loan_data (1).csv")
test_df = pd.read_csv("test_loan_data (1).csv")
```

In [3]:
```
train_df.shape
```

Out[3]: (80000, 28)

In [4]:
```
test_df.shape
```

Out[4]: (20000, 27)

In [5]:
```
pd.set_option('display.max_columns',29)
```

In [6]:
```
df = train_df.copy()
df.head()
```

Out[6]:

ıl	open_acc	pub_rec	pub_rec_bankruptcies	purpose	revol_bal	revol_util	sub_grade	term	
0	7	0	0.0	debt_consolidation	5338	93.6	E1	60 months	cc
0	5	0	0.0	debt_consolidation	19944	60.3	B1	36 months	
0	7	0	0.0	debt_consolidation	23199	88.5	B5	36 months	cc
0	12	0	0.0	debt_consolidation	18425	69.0	B2	36 months	cc
0	23	0	0.0	debt_consolidation	34370	90.0	F5	60 months	Cor

In [7]:
```
df.shape
```

Out[7]:
```
(80000, 28)
```

3: Dataset Overview

⬆️ Table Contents

In [8]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 80000 entries, 0 to 79999
Data columns (total 28 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   addr_state            80000 non-null  object 
 1   annual_inc            80000 non-null  float64
 2   earliest_cr_line      80000 non-null  object 
 3   emp_length            75412 non-null  object 
 4   emp_title             74982 non-null  object 
 5   fico_range_high       80000 non-null  int64  
 6   fico_range_low        80000 non-null  int64  
 7   grade                 80000 non-null  object 
 8   home_ownership        80000 non-null  object 
 9   application_type      80000 non-null  object 
 10  initial_list_status   80000 non-null  object 
 11  int_rate              80000 non-null  float64
 12  loan_amnt             80000 non-null  int64  
 13  num_actv_bc_tl        76052 non-null  float64
 14  mort_acc              77229 non-null  float64
 15  tot_cur_bal           76052 non-null  float64
 16  open_acc              80000 non-null  int64  
 17  pub_rec               80000 non-null  int64  
 18  pub_rec_bankruptcies  79969 non-null  float64
 19  purpose               80000 non-null  object 
 20  revol_bal             80000 non-null  int64  
 21  revol_util            79947 non-null  float64
 22  sub_grade             80000 non-null  object 
 23  term                  80000 non-null  object 
 24  title                 79030 non-null  object 
 25  total_acc             80000 non-null  int64  
 26  verification_status   80000 non-null  object 
 27  loan_status           80000 non-null  object 
dtypes: float64(7), int64(7), object(14)
memory usage: 17.1+ MB
```

Inference:

- The dataset contains **80000** entries.
- Each entry represents a loan application.
- There are **28** columns in the dataset.
- The columns represent various features such as:
  - State of the borrower's address (addr_state)
  - Annual income (annual_inc)
  - Earliest reported credit line (earliest_cr_line)
  - Employment length (emp_length)
  - Job title (emp_title)
  - FICO score range (fico_range_high, fico_range_low)
  - Loan grade (grade)

    - Home ownership status (home_ownership)
    - Application type (application_type)
    - Initial listing status (initial_list_status)
    - Interest rate (int_rate)
    - Loan amount (loan_amnt)
    - Number of active bankcard accounts (num_actv_bc_tl)
    - Number of mortgage accounts (mort_acc)
    - Total current balance of all accounts (tot_cur_bal)
    - Number of open credit lines (open_acc)
    - Number of derogatory public records (pub_rec)
    - Number of public record bankruptcies (pub_rec_bankruptcies)
    - Loan purpose (purpose)
    - Total credit revolving balance (revol_bal)
    - Revolving line utilization rate (revol_util)
    - Loan subgrade (sub_grade)
    - Term of the loan (term)
    - Loan title (title)
    - Total number of credit lines (total_acc)
    - Income verification status (verification_status)
    - Loan status (loan_status)
    - Loan type (type)
- There are missing values in several columns such as:
    - Employment length (emp_length)
    - Job title (emp_title)
    - Number of active bankcard accounts (num_actv_bc_tl)
    - Number of mortgage accounts (mort_acc)
    - Total current balance of all accounts (tot_cur_bal)
    - Number of public record bankruptcies (pub_rec_bankruptcies)
    - Revolving line utilization rate (revol_util)
    - Loan title (title)
    - Loan status (loan_status)
- The target variable is '**loan_status**', which represents the status of the loan application.
- The features include a mix of numerical (float64) and categorical (object) data types.
- The dataset requires handling missing values before proceeding with analysis or modeling.
- Further analysis could involve exploring relationships between different features and the loan status, as well as building predictive models to classify loan applications based on their status.

Note 📝:
Based on the data types and the feature explanations provided earlier, we identified that 3 columns (**pub_rec**,**mort_acc**, **pub_rec_bankruptcies**) are categorical in terms of their semantics. These features must have string (object) data type to ensure proper analysis and interpretation in subsequent steps:

In [9]:
```
categorical_columns_semantics = ['fico_range_high','fico_range_low','num_actv_bc_tl','


# Convert these columns to string (object) data type
for column in categorical_columns_semantics:
    if column in df.columns:
        df[column] = df[column].astype(str)
        train_df[column] = train_df[column].astype(str)
        test_df[column] = test_df[column].astype(str)

# Verify the changes in data types
df.dtypes
```

Out[9]:
```
addr_state              object
annual_inc             float64
earliest_cr_line        object
emp_length              object
emp_title               object
fico_range_high         object
fico_range_low          object
grade                   object
home_ownership          object
application_type        object
initial_list_status     object
int_rate               float64
loan_amnt                int64
num_actv_bc_tl          object
mort_acc                object
tot_cur_bal            float64
open_acc                object
pub_rec                 object
pub_rec_bankruptcies    object
purpose                 object
revol_bal                int64
revol_util             float64
sub_grade               object
term                    object
title                   object
total_acc               object
verification_status     object
loan_status             object
dtype: object
```

3.1: Summary Statistics for Numerical Features

⬆️ Table Contents

In [10]:
```
df.describe().T
```

Out[10]:

	count	mean	std	min	25%	50%	75%	max
annual_inc	80000.0	76046.143138	69020.055377	0.00	46000.00	65000.00	90000.00	7141778.00
int_rate	80000.0	13.232898	4.771705	5.31	9.75	12.74	15.99	30.99
loan_amnt	80000.0	14403.867813	8703.826298	750.00	7925.00	12000.00	20000.00	40000.00
tot_cur_bal	76052.0	141586.358991	159371.366632	0.00	29642.00	81000.50	211027.25	5172185.00
revol_bal	80000.0	16289.340975	22649.147472	0.00	5965.75	11111.00	19635.00	1023940.00
revol_util	79947.0	51.899142	24.504836	0.00	33.50	52.20	70.80	152.60

Inference:

- **Annual Income:**

- The average annual income of loan applicants is approximately **$76,046**, with a considerable standard deviation of **$69020**. This indicates a wide range of income levels among applicants.
- The minimum reported income is **$0**, which might be an anomaly or could represent unemployed individuals or students or those with no reported income.
- The maximum reported income is significantly high at **$7,141,778**, which could represent outliers or high-income individuals.

- **Interest Rate:**
  - The average interest rate on loans is approximately **13.23%**, with a standard deviation of **4.77%**. This signifies the range of interest rates offered to borrowers.
  - Interest rates range from **5.31%** to **30.99%**, with most falling between **9.75%** and **15.99%**. This indicates variability in loan pricing based on factors like creditworthiness and market conditions.
- **Loan Amount:**
  - The average loan amount requested is **$14,404**, with a standard deviation of **$8,704**. This suggests variability in loan sizes among applicants.
  - Loan amounts range from **$750** to **$40,000**, with the majority between **$7,925** and **$20,000**.
- **Other Numerical Features:**
  - Features like'**revol_bal**', and '**revol_util**', also exhibit variability in their distributions, indicating diversity among applicants in terms of their credit and financial profiles.

3.2: Summary Statistics for Categorical Features

⬆️ Table Contents

In [11]:
```
df.describe(include="object")
```

Out[11]:

	addr_state	earliest_cr_line	emp_length	emp_title	fico_range_high	fico_range_low	grade	home_
count	80000	80000	75412	74982	80000	80000	80000	
unique	51	640	11	36661	38	38	7	
top	CA	Sep-03	10+ years	Teacher	664	660	B	M
freq	11744	547	26278	1278	7267	7267	23502	

Inference:

- **Address State:** There are **51** unique states represented in the dataset, with **CA** being the most frequent state, occurring **11744** times. This indicates that a significant number of loan applicants are from California.
- **Earliest Credit Line:** The feature represents the month and year when the borrower's earliest reported credit line was opened. It has **640** unique values, with the most common being **Sep-03**, occurring **547** times.
- **Employment Length:** This feature indicates the length of employment in years. The most frequent value is **10+ years**, occurring **26278** times, suggesting that many applicants have long-term employment.
- **Employment Title:** There are **36661** unique employment titles, with **Teacher** being the most common, occurring **1278** times. This indicates a wide variety of job titles among loan applicants.
- **FICO Score Range:** The FICO score range reported by applicants ranges from **660** to **850**, with the average high score being **664** and the average low score being **660**.
- **Loan Grade:** The loan grade assigned by the lending institution ranges from **A** to **G**, with **B** being the most common grade, occurring **23502** times.
- **Home Ownership:** There are **6** unique categories representing home ownership status, with **MORTGAGE** being the most common, occurring **39628** times.

- **Application Type:** Indicates whether the loan application is individual or joint. The majority of applications are **Individual**, occurring **78446** times.
- **Initial List Status:** Represents the initial listing status of the loan. The majority of loans have an initial list status of **W**, occurring **46745** times.
- **Number of Active Bankcard Accounts:** There are **29** active bankcard accounts reported by applicants, with most having **3** active accounts.
- **Number of Mortgage Accounts:** There are **29** Mortgage Accounts accounts reported by applicants, with most having **0** mortgage accounts.
- **Number of Open Credit Lines:** There are **56** open credit lines reported by applicants, with most having **9** open credit lines.
- **Number of Public Record Items:** There are **15** public record items reported by applicants, with most having **0** public record items.
- **Number of Public Record Bankruptcies:** There are **9** Public Record Bankruptcies reported by applicants, with most having **0** Bankruptcies.
- **Purpose:** Describes the purpose of the loan request. The most common purpose is **Debt consolidation**, occurring **46418** times.
- **Loan Sub Grade:** Represents a sub-category within the loan grade, ranging from **A1** to **G5**. The most common sub-grade is **C1**, occurring **4982** times.
- **Term:** Indicates the number of payments on the loan, either **36 months** or **60 months**. The majority of loans have a term of **36 months**, occurring **60750** times.
- **Title:** The loan title provided by the borrower. There are **5348** unique titles, with **Debt consolidation** being the most common, occurring **39396** times.
- **Total Number of Credit Lines:** The average total number of credit lines reported by applicants is **22**, with a maximum of **107** credit lines.
- **Verification Status:** Indicates whether income was verified by the lending institution. The most common verification status is **Source Verified**, occurring **30855** times.
- **Loan Status:** Represents the status of the loan. It appears that **paid** is the most common status, occurring **64030** times.which will lead to unbalance of data

4: EDA

⬆️ Table Contents

Note 📝:

It is better to perform Exploratory Data Analysis (EDA) before handling missing values. Doing EDA after handling missing values have a higher likelihood of using techniques based on false assumptions, which will hamper data quality. Poorer data quality will naturally have a negative influence on any machine learning model built in the subsequent modeling phase (garbage in, garbage out).

Using EDA for Missing Data:

- EDA is an effective way to grasp the key characteristics of the dataset and its missing values.
- By arming yourself with more information on the missing data, you will be able to make more informed decisions with regard to how these missing values should be dealt with.

In [12]:
```
# Set the resolution of the plotted figures
plt.rcParams['figure.dpi'] = 200

# Configure Seaborn plot styles: Set background color and use dark grid
sns.set(rc={'axes.facecolor': '#faded9'}, style='darkgrid')
```

4.1: Univariate Analysis

⬆️ Table Contents

4.1.1 Numerical Features Univariate Analysis

⬆️ Table Contents

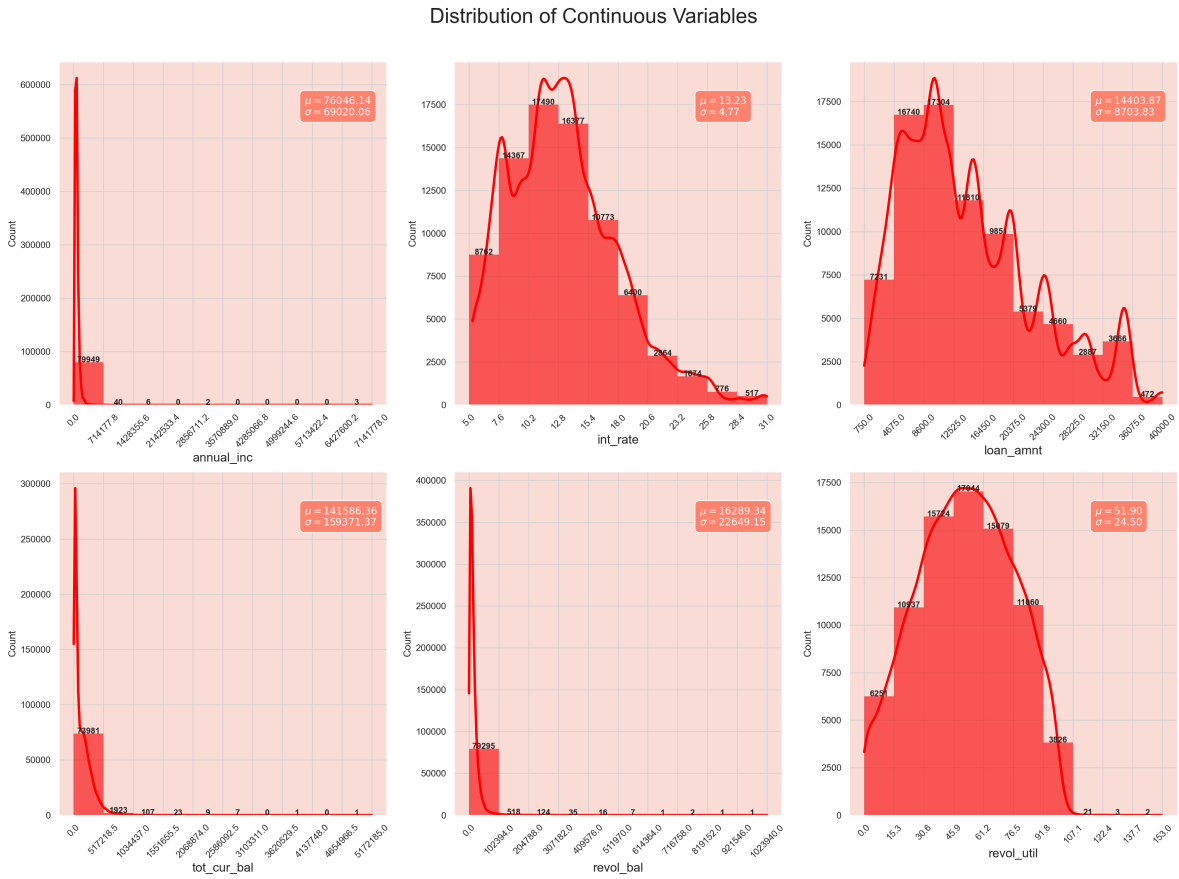
In [13]:
```
continuous_features = df.describe().columns
```

In [14]:
```
continuous_features
```

Out[14]:
```
Index(['annual_inc', 'int_rate', 'loan_amnt', 'tot_cur_bal', 'revol_bal',
       'revol_util'],
      dtype='object')
```

In [15]:
```
df_continuous = df[continuous_features]
```

In [16]:
```
# Set up the subplot
fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(20, 15))

# Loop to plot histograms for each continuous feature
for i, col in enumerate(df_continuous.columns):
    x = i // 3
    y = i % 3
    values, bin_edges = np.histogram(df_continuous[col],
                                     range=(np.floor(df_continuous[col].min()), np.cei

    graph = sns.histplot(data=df_continuous, x=col, bins=bin_edges, kde=True, ax=ax[x,
                         edgecolor='none', color='red', alpha=0.6, line_kws={'lw': 3})
    ax[x, y].set_xlabel(col, fontsize=15)
    ax[x, y].set_ylabel('Count', fontsize=12)
    ax[x, y].set_xticks(np.round(bin_edges, 1))
    ax[x, y].set_xticklabels(ax[x, y].get_xticks(), rotation=45)
    ax[x, y].grid(color='lightgrey')

    for j, p in enumerate(graph.patches):
        ax[x, y].annotate('{}'.format(p.get_height()), (p.get_x() + p.get_width() / 2,
                          ha='center', fontsize=10, fontweight="bold")

    textstr = '\n'.join((
        r'$\mu=%.2f$' % df_continuous[col].mean(),
        r'$\sigma=%.2f$' % df_continuous[col].std()
    ))
    ax[x, y].text(0.75, 0.9, textstr, transform=ax[x, y].transAxes, fontsize=12, verti
                  color='white', bbox=dict(boxstyle='round', facecolor='#ff826e', edge

# ax[3,2].axis('off')
plt.suptitle('Distribution of Continuous Variables', fontsize=25)
plt.tight_layout()
plt.subplots_adjust(top=0.92)
plt.show()
```



Inference:

- **Annual Income (annual_inc):**
  - Right-skewed distribution with a peak at zero(Pareto distribution), suggesting missing or invalid income data.
  - Average reported annual income: **$76046**, standard deviation: **$69020**.
  - Understanding income variability is crucial for assessing financial stability and repayment capacity.
- **Interest Rate (int_rate):**
  - The distribution of interest rates shows that the majority of applicants have interest rates between **7.6%** and **15.4%**.
  - The average interest rate is **13.23%**, with a standard deviation of **4.77%**, indicating some variability in interest rates among loans.
- **Loan Amount:**
  - The distribution of loan amounts shows a decreasing sinusoidal pattern from around **$8600**.
  - The average loan amount is approximately **$14,404**, with a considerable standard deviation of **$8704**, indicating significant variability in loan amounts among applicants.
- **Total Current Balance (tot_cur_bal):**
  - The distribution of total current balances shows a high peak at 0 (Pareto distribution), indicating that a significant proportion of applicants have no current balances.
  - The presence of outliers, such as the maximum value of **$5,172,185**, suggests extreme values that deviate significantly from the typical range of balances.
  - The average total current balance is approximately **$141,586.4**, with a considerable standard deviation of **$159,371**, suggesting variability in total balances among applicants.
- **Revolving Balance (revol_bal):**
  - The distribution(Pareto distribution) of revolving balances shows a high peak increasing from 0 to around **$50,000**, indicating that a significant proportion of applicants have relatively low to moderate revolving balances.
  - The average revolving balance is approximately **$16,289**, with a considerable standard deviation of **$22,649**, suggesting significant variability in revolving balances among applicants.
- **Revolving Line Utilization Rate (revol_util):**
  - The distribution of revolving line utilization rates shows a somewhat normal distribution.
  - The average revolving line utilization rate is approximately **51.93%**, with a standard deviation of **24.5%**, indicating variability in the utilization rates among applicants.

4.1.2: Categorical Features Univariate Analysis

⬆️ Table Contents

4.1.2.1: Low Cardinality Categorical Features

⬆️ Table Contents

In [17]:
```
# Filter low cardinality features for the univariate analysis, excluding 'type' column
low_cardinality_fea = ['application_type','grade','home_ownership','initial_list_statu
df_low_cardinality = df[low_cardinality_fea]
```

In [18]:
```
# Set up the subplot
fig, ax = plt.subplots(nrows=4, ncols=3, figsize=(20, 18))

# Loop to plot bar charts for each categorical feature in the 4x2 layout
for i, col in enumerate(low_cardinality_fea):
    row = i // 3
    col_idx = i % 3

    # Calculate frequency percentages
    value_counts = df[col].value_counts(normalize=True).mul(100).sort_values()

    # Plot bar chart
    value_counts.plot(kind='barh', ax=ax[row, col_idx], width=0.8, color='red')

    # Add frequency percentages to the bars
    for index, value in enumerate(value_counts):
        ax[row, col_idx].text(value, index, str(round(value, 1)) + '%', fontsize=15, w

    ax[row, col_idx].set_xlim([0, 95])
    ax[row, col_idx].set_xlabel('Frequency Percentage', fontsize=12)
    ax[row, col_idx].set_title(f'{col}', fontsize=20)

ax[3,2].axis('off')
plt.suptitle('Distribution of low Cardinality Categorical Variables', fontsize=22)
plt.tight_layout()
plt.subplots_adjust(top=0.95)
plt.show()
```

Inference:

- **Application Type:** The majority of loan applications (98.1%) are from individuals rather than joint applications, indicating that individual borrowers are the primary applicants.
- **Grade Distribution:** Grades "B" and "C" are the most common grades among loan applicants, suggesting that a significant proportion of applicants fall within these credit risk categories.
- **Home Ownership:** The highest proportion of applicants indicate 'Mortgage' as their home ownership status, indicating that a substantial number of borrowers are homeowners with existing mortgages.
- **Initial List Status:** The majority of loans have an initial listing status of 'W' (Whole), suggesting that most loans are initially listed as a whole rather than fractional.
- **Loan Term:** The most preferred loan term among applicants is 36 months, indicating that a significant proportion of borrowers opt for shorter loan durations.
- **Verification Status:** The majority of loan applicants have their income source as 'Source verified', with a somewhat balanced distribution between 'Verified' and 'Not Verified' statuses.
- **Loan Status:** The most common loan status among applicants is 'Paid', indicating that a large number of borrowers have successfully repaid their loans.
- **Employment Length:** Applicants with an employment length of '10+ years' are the most prevalent, indicating a significant number of borrowers with long-term employment.
- **Purpose:** The most common purpose for loan applications is 'Debt Consolidation', followed by 'Credit Card', suggesting that many borrowers seek loans to consolidate existing debts or manage credit card balances.
- **pub_rec:** most of the applicants have '0' and '1' Public Record items
- **pub_rec_bankruptcies:** most of the applicants have '0' and '1' Public Record Bankruptcies

4.1.2.2: High Cardinality Categorical Features

⬆️ Table Contents

WE will focus on only TOP10 from each one

In [19]:
```
categorical_features = df.columns.difference(continuous_features)
```

In [20]:
```
# Filter high cardinality features for the univariate analysis, excluding 'type' colum
high_cardinality_fea = list(set(categorical_features) - set(low_cardinality_fea))
df_high_cardinality = df[high_cardinality_fea]
```

In [21]:

```
# Set up the subplot
fig, ax = plt.subplots(nrows=4, ncols=3, figsize=(20, 30))

# Loop to plot bar charts for each categorical feature in the 4x2 layout
for i, col in enumerate(high_cardinality_fea):
    row = i // 3
    col_idx = i % 3

    # Calculate frequency percentages
    value_counts = df[col].value_counts(normalize=True).mul(100).sort_values(ascending

    # Plot bar chart
    value_counts.plot(kind='barh', ax=ax[row, col_idx], width=0.8, color='red')

    # Add frequency percentages to the bars
    for index, value in enumerate(value_counts):
        ax[row, col_idx].text(value, index, str(round(value, 1)) + '%', fontsize=15, w

    ax[row, col_idx].set_xlim([0, 95])
    ax[row, col_idx].set_xlabel('Frequency Percentage', fontsize=12)
    ax[row, col_idx].set_title(f'{col}', fontsize=20)

ax[3,2].axis('off')

plt.suptitle('Distribution of High Cardinality Categorical Variables', fontsize=22)
plt.tight_layout()
plt.subplots_adjust(top=0.9)
plt.show()
```

Distribution of High Cardinality Categorical Variables

Inference:

- **FICO range high:** The FICO range high scores from 664 to 709 dominate, with 664 being the most prevalent at 9.1%, gradually decreasing thereafter.
- **Active bankcard accounts (num_actv_bc_tl):** Approximately 20.2% of applicants have 3 active bankcard accounts, followed by 2 active accounts at 19.6%.

- **Sub-Grade:** The most prevalent sub-grade is C1, constituting 6.3% of the total, followed closely by B4 and B5, each comprising approximately 6.2% and 6.1%, respectively. Sub-grades from the 'B' and 'C' categories dominate the top positions, with slight variations in percentages
- **Total number of credit lines:** The most common count is 22, representing 3.6% of the dataset, followed closely by 23 and 19 at approximately 3.6% each
- **Title:** The most common title for loan applications is 'Debt Consolidation', followed by 'Credit Card Refinancing', indicating that many borrowers seek loans for debt consolidation purposes or to refinance existing credit card balances.
- **mortgage accounts:** The most prevalent count is 0, representing 38.8% of the dataset, followed by 1 at 16.9% and 2 at 13.9%
- **State:** The state with the highest number of loan applicants is 'CA' (California), followed by 'TX' (Texas), indicating that these states have the highest demand for loans among borrowers.
- **FICO range low:** The FICO range low scores from 660 to 705 dominate, with 600 being the most prevalent at 9.1%, gradually decreasing thereafter.
- **Employment Title:** The most common employment title among loan applicants is 'Teacher', followed by 'Manager', suggesting that individuals in these professions are more likely to apply for loans.
- **Open credit lines (open_acc):** The most common count is 9, comprising 9.0% of the dataset, followed by 10 at 8.5% and 8 at 8.4%
- **Earliest Credit Line:** The distribution of earliest credit line dates shows that around 0.5% of applicants have their earliest credit line in Months from August - November, indicating a common occurrence for this specific time period among loan applicants

4.2: Bivariate Analysis

⬆️ Table Contents

> **Note** 📝**:**
>
> For bivariate analysis, we will utilize the `train_df` dataframe as the `test_df` dataframe lacks a target feature. This ensures consistency in the analysis and modeling process.

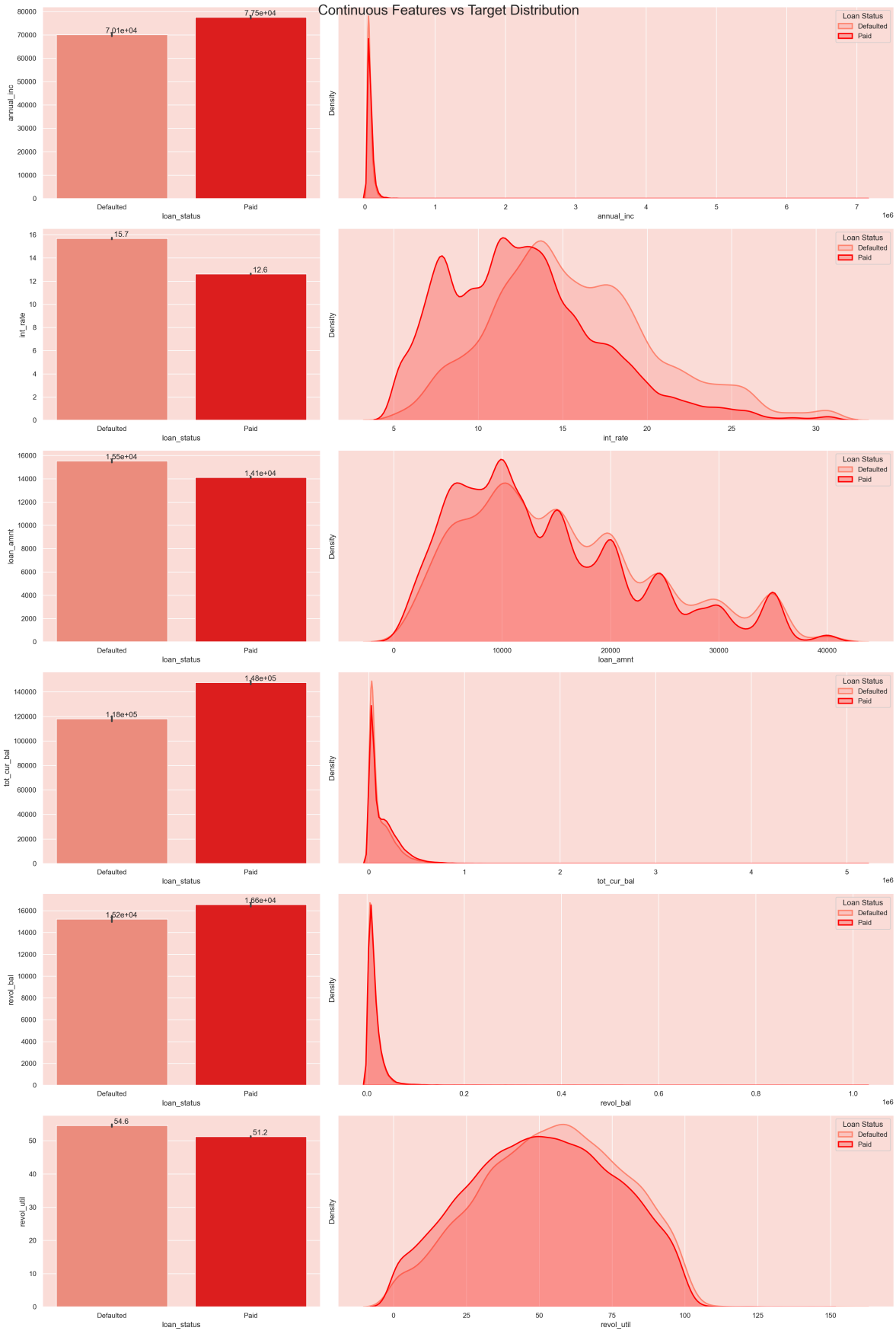
4.2.1: Numerical Features vs Target (loan_status)

⬆️ Table Contents

In [22]:
```
continuous_features = train_df.describe().columns
tarin_df_continuous = train_df[continuous_features]
```

In [23]:

```
# Set color palette
sns.set_palette(['#ff826e', 'red'])

# Create the subplots
fig, ax = plt.subplots(len(continuous_features), 2, figsize=(20,30), gridspec_kw={'wid

# Loop through each continuous feature to create barplots and kde plots
for i, col in enumerate(continuous_features):
    # Barplot showing the mean value of the feature for each target category
    graph = sns.barplot(data=train_df, x="loan_status", y=col, ax=ax[i,0])

    # KDE plot showing the distribution of the feature for each target category
    sns.kdeplot(data=train_df[train_df["loan_status"]=='Defaulted'], x=col, fill=True,
    sns.kdeplot(data=train_df[train_df["loan_status"]=='Paid'], x=col, fill=True, line
    ax[i,1].set_yticks([])
    ax[i,1].legend(title='Loan Status', loc='upper right')

    # Add mean values to the barplot
    for cont in graph.containers:
        graph.bar_label(cont, fmt='         %.3g')

# Set the title for the entire figure
plt.suptitle('Continuous Features vs Target Distribution', fontsize=22)
plt.tight_layout()
plt.show()
```



Inference:

- The bivariate analysis conducted with respect to the target feature (`loan_status`) reveals notable differences in the distributions of certain continuous features between the "Paid" and "Default" categories.
- Features such as **int_rate**, **loan_amount**, and **revol_util** exhibit higher mean for the "Defaulted" category compared to the "paid" category.

- This suggests that these features may have a significant impact on the likelihood of a loan defaulting, warranting further investigation and consideration in predictive modeling and risk assessment.

4.2.2: Categorical Features vs Target (loan_status)

⬆️ Table Contents

4.2.2.1: Low Cardinality Categorical Features vs Target (loan_status)

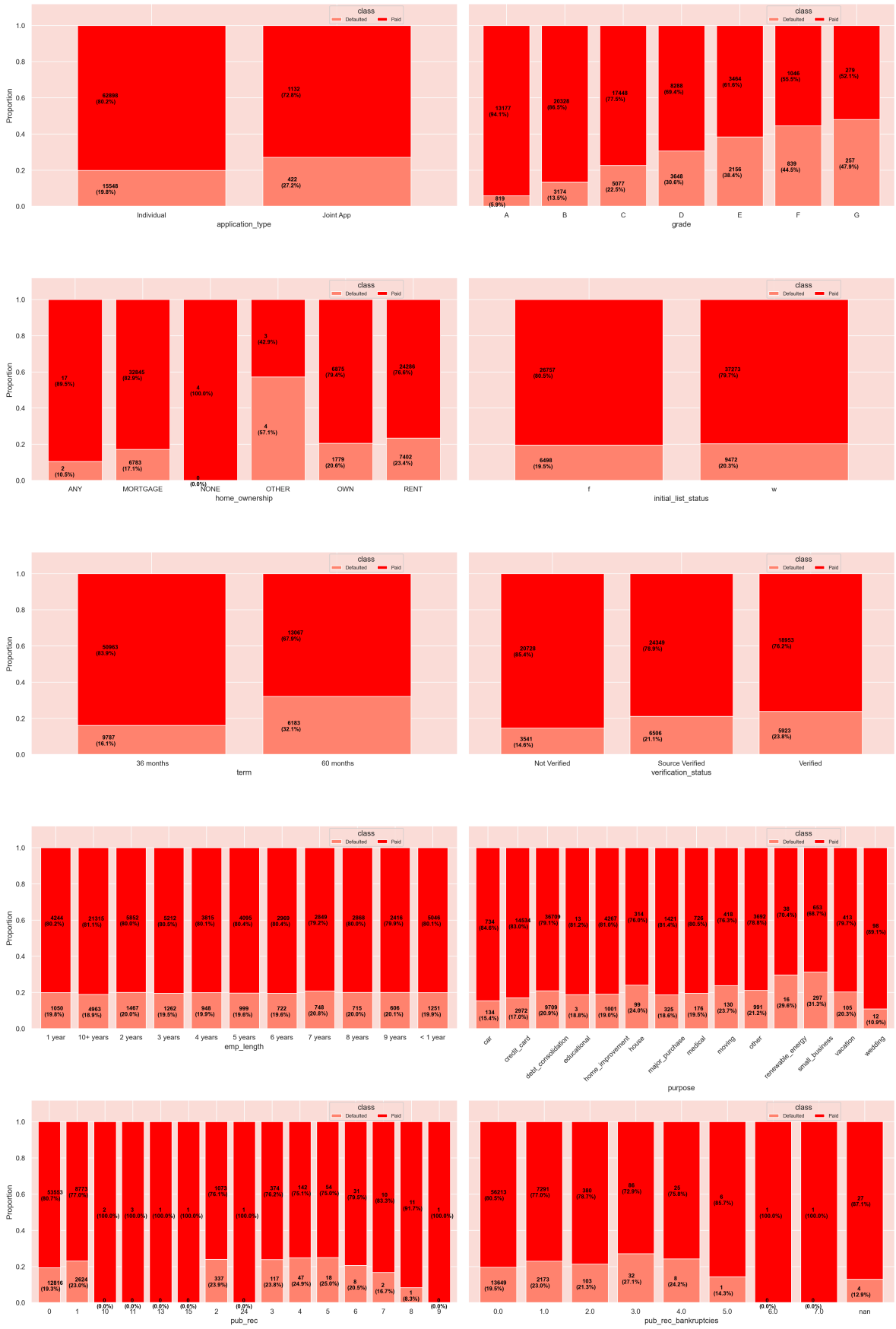
⬆️ Table Contents

In [24]:
```
# Remove 'loan_status' from the low_cardinality_features
low_cardinality_fea = [feature for feature in low_cardinality_fea if feature != 'loan_

low_cardinality_fea
```

Out[24]:
```
['application_type',
 'grade',
 'home_ownership',
 'initial_list_status',
 'term',
 'verification_status',
 'emp_length',
 'purpose',
 'pub_rec',
 'pub_rec_bankruptcies']
```

In [25]:
```
# Calculate the number of required subplots
num_plots = len(low_cardinality_fea)
num_rows = (num_plots + 1) // 2  # Add one extra row if the number of plots is odd

# Set up the subplot
fig, ax = plt.subplots(nrows=num_rows, ncols=2, figsize=(20, num_rows * 6))

for i, col in enumerate(low_cardinality_fea):
    # Calculate the row and column index
    x, y = i // 2, i % 2

    # Create a cross tabulation showing the proportion of purchased and non-purchased
    cross_tab = pd.crosstab(index=train_df[col], columns=train_df['loan_status'])

    # Using the normalize=True argument gives us the index-wise proportion of the data
    cross_tab_prop = pd.crosstab(index=train_df[col], columns=train_df['loan_status'],

    # Define colormap
    cmp = ListedColormap(['#ff826e', 'red'])

    # Plot stacked bar charts
    cross_tab_prop.plot(kind='bar', ax=ax[x, y], stacked=True, width=0.8, colormap=cmp
                        legend=False, ylabel='Proportion', sharey=True)

    # Add the proportions and counts of the individual bars to our plot
    for idx, val in enumerate([*cross_tab.index.values]):
        for (proportion, count, y_location) in zip(cross_tab_prop.loc[val],cross_tab.l
            ax[x, y].text(x=idx-0.3, y=(y_location-proportion)+(proportion/2)-0.03,
                          s = f'    {count}\n({np.round(proportion * 100, 1)}%)',
                          color = "black", fontsize=9, fontweight="bold")

    # Add legend
    ax[x, y].legend(title='class', loc=(0.7,0.9), fontsize=8, ncol=2)
    # Set y limit
    ax[x, y].set_ylim([0,1.12])
    # Rotate xticks
    ax[x, y].set_xticklabels(ax[x, y].get_xticklabels(), rotation=0)

ax[3, 1].set_xticklabels(ax[3, 1].get_xticklabels(), rotation=45)
# Remove empty subplot if the number of plots is odd
if num_plots % 2 != 0:
    fig.delaxes(ax[num_rows-1, 1])

# Set title outside the subplots
plt.suptitle('low cardinality Categorical Features vs Loan Status(target) Stacked Barp
plt.tight_layout()
plt.show()
```

low cardinality Categorical Features vs Loan Status(target) Stacked Barplots



Inference:

- Upon analyzing the low cardinality categorical features, it's evident that they are directly associated with the loan status labeled as "Paid".
- For instance, the feature **home_ownership** with the category "NONE" exhibits all loan statuses as "Paid", indicating a potential bias in this category.

- Similarly, the feature **grade** with the category "G" shows somewhat equal proportions of loan statuses, which might suggest a different pattern compared to other grades.
- The feature **pub_rec** with the categories "10","9","10","11","13" and "15" exhibits all loan statuses as "Paid", indicating a potential bias in this category.
- Also,the feature **pub_rec_bankruptcies** with the categories "6" and "7" exhibits all loan statuses as "Paid", indicating a potential bias in this category.
- This direct association of certain categories with the "Paid" loan status hints at potential biasness in the dataset, which will be addressed in the subsequent data preprocessing and feature engineering steps to ensure fairness and model accuracy.

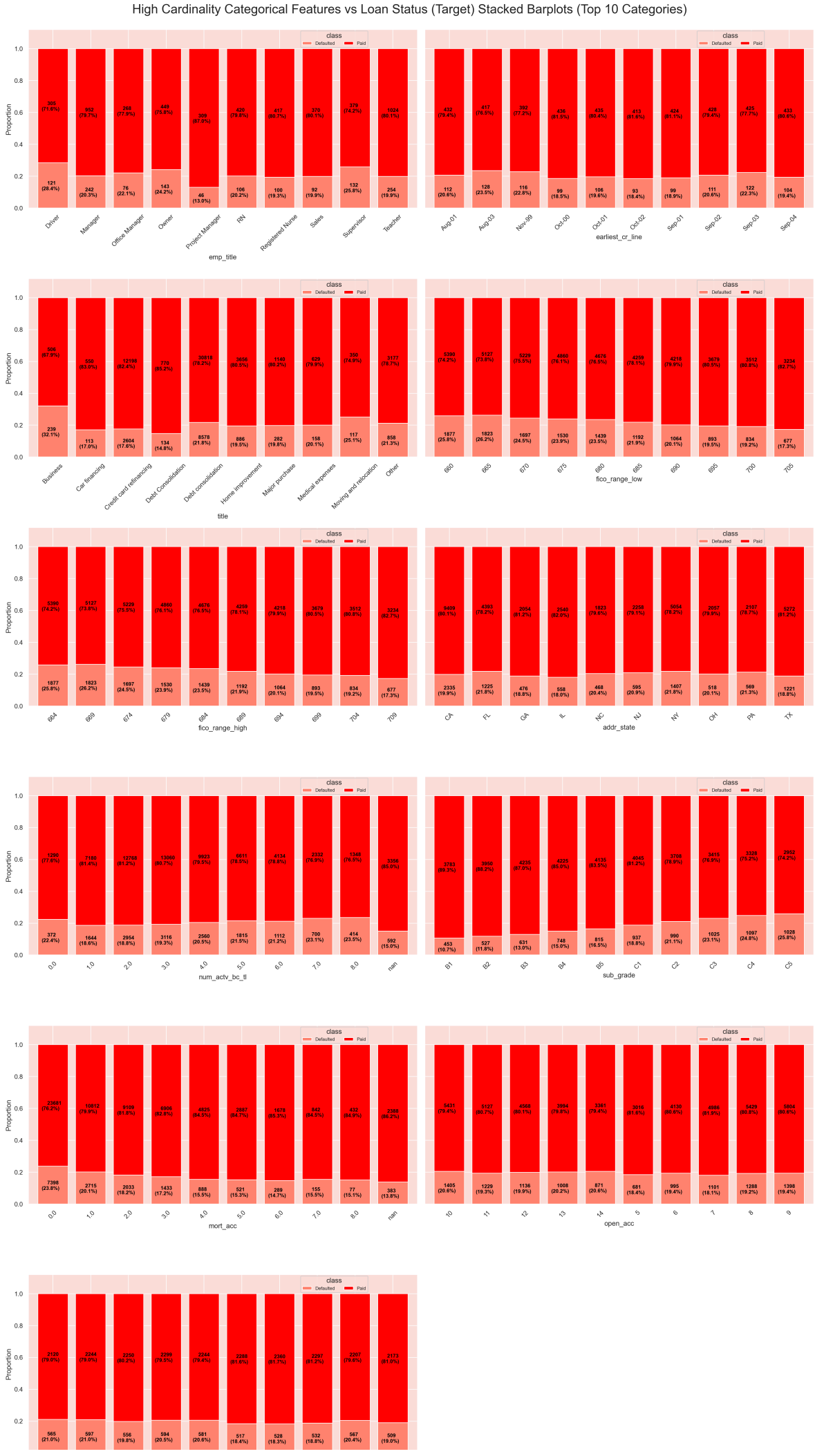
4.2.2.2: High Cardinality Categorical Features vs Target (loan_status)

⬆️ Table Contents

In [26]:
```
high_cardinality_fea
```

Out[26]:
```
['emp_title',
 'earliest_cr_line',
 'title',
 'fico_range_low',
 'fico_range_high',
 'addr_state',
 'num_actv_bc_tl',
 'sub_grade',
 'mort_acc',
 'open_acc',
 'total_acc']
```

In [27]:
```
# Calculate the number of required subplots
num_plots = len(high_cardinality_fea)
num_rows = (num_plots + 1) // 2  # Add one extra row if the number of plots is odd

# Set up the subplot
fig, ax = plt.subplots(nrows=num_rows, ncols=2, figsize=(20, num_rows * 6))

for i, col in enumerate(high_cardinality_fea):
    # Calculate the row and column index
    x, y = i // 2, i % 2

    # Get the top 10 most occurring categories
    top_categories = train_df[col].value_counts().head(10).index

    # Subset the dataframe to include only the top 10 categories
    subset_df = train_df[train_df[col].isin(top_categories)]

    # Create a cross tabulation showing the proportion of loan statuses for each categ
    cross_tab = pd.crosstab(index=subset_df[col], columns=subset_df['loan_status'])

    # Using the normalize=True argument gives us the index-wise proportion of the data
    cross_tab_prop = pd.crosstab(index=subset_df[col], columns=subset_df['loan_status'

    # Define colormap
    cmp = ListedColormap(['#ff826e', 'red'])

    # Plot stacked bar charts
    cross_tab_prop.plot(kind='bar', ax=ax[x, y], stacked=True, width=0.8, colormap=cmp
                        legend=False, ylabel='Proportion', sharey=True)

    # Add the proportions and counts of the individual bars to our plot
    for idx, val in enumerate([*cross_tab.index.values]):
        for (proportion, count, y_location) in zip(cross_tab_prop.loc[val],cross_tab.l
            ax[x, y].text(x=idx-0.3, y=(y_location-proportion)+(proportion/2)-0.03,
                          s = f'    {count}\n({np.round(proportion * 100, 1)}%)',
                          color = "black", fontsize=9, fontweight="bold")

    # Add legend
    ax[x, y].legend(title='class', loc=(0.7,0.9), fontsize=8, ncol=2)
    # Set y limit
    ax[x, y].set_ylim([0,1.12])
    # Rotate xticks
    ax[x, y].set_xticklabels(ax[x, y].get_xticklabels(), rotation=45)

# Remove empty subplot if the number of plots is odd
if num_plots % 2 != 0:
    fig.delaxes(ax[num_rows-1, 1])

# Set title outside the subplots
plt.suptitle('High Cardinality Categorical Features vs Loan Status (Target) Stacked Ba
plt.tight_layout()
plt.show()
```


High Cardinality Categorical Features vs Loan Status (Target) Stacked Barplots (Top 10 Categories)
class
Defaulted
Paid
Proportion
emp_title
Driver
Manager
Office Manager
Owner
Project Manager
RN
Registered Nurse
Sales
Supervisor
Teacher
earliest_cr_line
Aug-01
Aug-03
Nov-89
Oct-00
Oct-01
Oct-02
Sep-01
Sep-02
Sep-03
Sep-04
title
Business
Car financing
Credit card refinancing
Debt Consolidation
Debt consolidation
Home improvement
Major purchase
Medical expenses
Moving and relocation
Other
fico_range_low
660
665
670
675
680
685
690
695
700
705
fico_range_high
664
669
674
679
684
689
694
699
704
709
addr_state
CA
FL
GA
IL
NC
NJ
NY
OH
PA
TX
num_actv_bc_tl
0.0
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
nan
sub_grade
B1
B2
B3
B4
B5
C1
C2
C3
C4
C5
mort_acc
0.0
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
nan
open_acc
10
11
12
13
14
5
6
7
8
9

Inference:

- Upon analyzing the high cardinality categorical features, it's evident that they are directly associated with the loan status labeled as "Paid".
- This direct association of certain categories with the "Paid" loan status hints at potential biasness in the dataset, which will be addressed in the subsequent data preprocessing and feature engineering steps to ensure fairness and model accuracy.

5: Data Preprocessing

⬆️ Table Contents

5.1: Feature Engineering

⬆️ Table Contents

5.1.1: High Cardinal Categorical Features

⬆️ Table Contents

In [28]:
```
# Create a dataframe with the number of unique categories and missing values for each
high_cardinality_features = high_cardinality_fea
unique_categories = train_df[high_cardinality_features].nunique()
missing_values = train_df[high_cardinality_features].isnull().sum()

# Combine the two series into a single dataframe
high_cardinality_df = pd.DataFrame({'Unique_Categories': unique_categories, 'Missing_V

high_cardinality_df
```

Out[28]:

	Unique_Categories	Missing_Values
emp_title	36661	5018
earliest_cr_line	640	0
title	5348	970
fico_range_low	38	0
fico_range_high	38	0
addr_state	51	0
num_actv_bc_tl	29	0
sub_grade	35	0
mort_acc	29	0
open_acc	56	0
total_acc	107	0

Feature Engineering Actions:

- **Address State (addr_state):** Grouping the states of the USA into regions to provide a more generalized view of geographical location.
- **Earliest Credit Line (earliest_cr_line):** Converting categories into datetime format to simplify the data while retaining temporal information.
- **Employment Title (emp_title):** Removing this feature due to the large number of categories. However, it can be grouped into common domain roles or only the top N most frequent roles can be retained, with the rest grouped into an 'Other' category if needed in future work.
- **Sub Grade (sub_grade):** Dropping this feature as it is directly related to the grade feature, making it redundant and increasing data dimensionality.
- **Title:** Dropping this feature as it is directly related to the purpose feature, which would increase data dimensionality.
- **FICO Range Low and High:** we will create a new feature based on the FICO range's low and high values. The aim is to categorize FICO scores into different groups such as Poor, Fair, Good, and Excellent.
- **Mortgage Account:** Grouping all accounts with values greater than zero into 1.

By performing these feature engineering actions, we aim to simplify the dataset while retaining important information relevant to the loan domain. This will help in reducing the dimensionality of the data and improving the efficiency of subsequent analysis and modeling.

addr_state

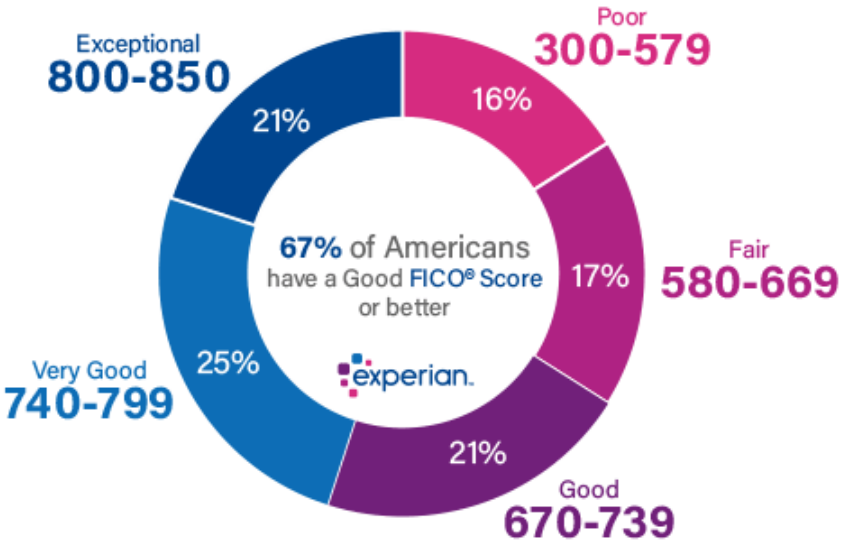
In [29]:
```
# Define mapping of states to regions
state_to_region = {
    'CT': 'Northeast',
    'ME': 'Northeast',
    'MA': 'Northeast',
    'NH': 'Northeast',
    'RI': 'Northeast',
    'VT': 'Northeast',
    'NJ': 'Northeast',
    'NY': 'Northeast',
    'PA': 'Northeast',
    'IL': 'Midwest',
    'IN': 'Midwest',
    'MI': 'Midwest',
    'OH': 'Midwest',
    'WI': 'Midwest',
    'IA': 'Midwest',
    'KS': 'Midwest',
    'MN': 'Midwest',
    'MO': 'Midwest',
    'NE': 'Midwest',
    'ND': 'Midwest',
    'SD': 'Midwest',
    'DE': 'South',
    'FL': 'South',
    'GA': 'South',
    'MD': 'South',
    'NC': 'South',
    'SC': 'South',
    'VA': 'South',
    'DC': 'South',
    'WV': 'South',
    'AL': 'South',
    'KY': 'South',
    'MS': 'South',
    'TN': 'South',
    'AR': 'South',
    'LA': 'South',
    'OK': 'South',
    'TX': 'South',
    'AZ': 'West',
    'CO': 'West',
    'ID': 'West',
    'MT': 'West',
    'NV': 'West',
    'NM': 'West',
    'UT': 'West',
    'WY': 'West',
    'AK': 'West',
    'CA': 'West',
    'HI': 'West',
    'OR': 'West',
    'WA': 'West'
}

# Map states to regions and drop addr_state column
train_df['region'] = train_df['addr_state'].map(state_to_region)
train_df.drop(columns=['addr_state'], inplace=True)

test_df['region'] = test_df['addr_state'].map(state_to_region)
test_df.drop(columns=['addr_state'], inplace=True)
```

earliest_cr_line

In [30]:
```
# Convert 'earliest_cr_line' to datetime format
train_df['earliest_cr_line'] = pd.to_datetime(train_df['earliest_cr_line'], format='%b

# Now you can access the year component using the .dt accessor
train_df['earliest_cr_line'] = train_df['earliest_cr_line'].dt.year
```

In [31]:
```
# Convert 'earliest_cr_line' to datetime format
test_df['earliest_cr_line'] = pd.to_datetime(test_df['earliest_cr_line'], format='%b-%

# Now you can access the year component using the .dt accessor
test_df['earliest_cr_line'] = test_df['earliest_cr_line'].dt.year
```

fico_range_high & fico_range_low



In [32]:
```
# Convert 'fico_range_low' and 'fico_range_high' to numerical type
train_df['fico_range_low'] = train_df['fico_range_low'].astype(float)
train_df['fico_range_high'] = train_df['fico_range_high'].astype(float)

# Calculate average FICO score
train_df['avg_fico_score'] = (train_df['fico_range_low'] + train_df['fico_range_high']

# Create bins and labels for FICO score ranges
bins = [299, 579, 669, 739, 799, 850]
labels = ['Poor', 'Fair', 'Good', 'Very Good', 'Exceptional']

# Bin the average FICO score
train_df['fico_score_category'] = pd.cut(train_df['avg_fico_score'], bins=bins, labels
```

In [33]:
```

test_df['fico_range_low'] = test_df['fico_range_low'].astype(float)
test_df['fico_range_high'] = test_df['fico_range_high'].astype(float)

# Calculate average FICO score
test_df['avg_fico_score'] = (test_df['fico_range_low'] + test_df['fico_range_high']) /

test_df['fico_score_category'] = pd.cut(test_df['avg_fico_score'], bins=bins, labels=l
```

In [34]:
```
# Drop the 'avg_fico_score','fico_range_low','fico_range_high' columns, no longer need
train_df.drop(columns=['avg_fico_score','fico_range_low','fico_range_high'],axis=1, in
test_df.drop(columns=['avg_fico_score','fico_range_low','fico_range_high'],axis=1, inp
```

mort_acc

In [35]:
```
def mort_acc(number):
    if number == 0.0:
        return 0
    elif number >= 1.0:
        return 1
    else:
        return number
```

In [36]:
```
train_df['mort_acc'] = train_df['mort_acc'].astype(float)
train_df['mort_acc'] = train_df.mort_acc.apply(mort_acc)
```

In [37]:
```
test_df['mort_acc'] = test_df['mort_acc'].astype(float)
test_df['mort_acc'] = test_df.mort_acc.apply(mort_acc)
```

sub_grade, title, emp_title

In [38]:
```
train_df.drop(columns=['sub_grade','title','emp_title'],axis=1, inplace=True)
test_df.drop(columns=['sub_grade','title','emp_title'],axis=1, inplace=True)
```

5.1.2: Low Cardinal Categorical Features

⬆️ Table Contents

Feature Engineering Actions:

- **home_ownership:** We notice a direct association between the "NONE" category and the "paid" category in the target feature. To address this, we will group "NONE," "ANY," and "OTHER" into an "Other" category.
- **term:** We observe two unique categories: 36 months and 60 months. To align with its semantics, we will convert this feature into numeric values representing the respective term lengths.
- **emp_length:** Based on bivariate analysis, emp_length exhibits significant bias towards the "paid" category in the target feature. Consequently, we opt to drop this feature. However, we may consider grouping it into "< 1 year" and "10+ years" for future analysis, pending discussion with business stakeholders regarding its relevance in the loan domain.
- **purpose:** We will group the categories based on their specific domain knowledge and requirements.
- **Public Record:** Grouping all records with values greater than zero into 1.
- **Public Record Bankruptcies:** Grouping all records with values greater than zero into 1.

By performing these feature engineering actions, we aim to simplify the dataset while retaining important information relevant to the loan domain. This will help in reducing the dimensionality of the data and improving the efficiency of subsequent analysis and modeling.

In [39]:
```
low_cardinality_fea
```

Out[39]:
```
['application_type',
 'grade',
 'home_ownership',
 'initial_list_status',
 'term',
 'verification_status',
 'emp_length',
 'purpose',
 'pub_rec',
 'pub_rec_bankruptcies']
```

home_ownership

In [40]:
```
train_df['home_ownership'].unique()
```

Out[40]:
```
array(['MORTGAGE', 'RENT', 'OWN', 'ANY', 'NONE', 'OTHER'], dtype=object)
```

In [41]:
```
train_df['home_ownership'] = train_df['home_ownership'].replace(['ANY', 'NONE', 'OTHER
test_df['home_ownership'] = test_df['home_ownership'].replace(['ANY', 'NONE', 'OTHER']
```

term

In [42]:
```
train_df['term'].unique()
```

Out[42]:
```
array([' 60 months', ' 36 months'], dtype=object)
```

In [43]:
```
train_df['term'] = train_df['term'].str.strip()
test_df['term'] = test_df['term'].str.strip()
```

In [44]:
```
# Define the mapping
term_mapping = {'36 months': 36, '60 months': 60}

# Apply the mapping using the replace function
train_df['term'] = train_df['term'].replace(term_mapping)
test_df['term'] = test_df['term'].replace(term_mapping)
```

purpose

In [45]:
```
train_df['purpose'].nunique()
```

Out[45]:
```
14
```

In [46]:
```
# Map similar categories to new categories
category_mapping = {
    'credit_related': ['credit_card', 'debt_consolidation'],
    'specific_purpose': ['car', 'home_improvement', 'major_purchase', 'medical', 'hous
    'education_and_personal': ['educational', 'wedding', 'vacation', 'moving'],
    'business': ['small_business'],
    'other': ['renewable_energy', 'other']
}


# Function to map loan purpose to broad category
def map_purpose_to_category(purpose):
    for category, purposes in category_mapping.items():
        if purpose in purposes:
            return category
    return 'Unknown'  # Default category if purpose doesn't match any
```

In [47]:
```
train_df['purpose_category'] = train_df['purpose'].map(map_purpose_to_category)
```

In [48]:
```
test_df['purpose_category'] = test_df['purpose'].map(map_purpose_to_category)
```

In [49]:
```
train_df.drop('purpose', axis=1, inplace=True)
test_df.drop('purpose', axis=1, inplace=True)
```

pub_rec & pub_rec_bankruptcies

In [50]:
```
def pub_rec(number):
    if number == 0.0:
        return 0
    else:
        return 1

def pub_rec_bankruptcies(number):
    if number == 0.0:
        return 0
    elif number >= 1.0:
        return 1
    else:
        return number
```

In [51]:
```
train_df['pub_rec'] = train_df['pub_rec'].astype(float)
train_df['pub_rec'] = train_df.pub_rec.apply(pub_rec)

train_df['pub_rec_bankruptcies'] = train_df['pub_rec_bankruptcies'].astype(float)
train_df['pub_rec_bankruptcies'] = train_df.pub_rec_bankruptcies.apply(pub_rec_bankrup
```

In [52]:
```
test_df['pub_rec'] = test_df['pub_rec'].astype(float)
test_df['pub_rec'] = test_df.pub_rec.apply(pub_rec)

test_df['pub_rec_bankruptcies'] = test_df['pub_rec_bankruptcies'].astype(float)
test_df['pub_rec_bankruptcies'] = test_df.pub_rec_bankruptcies.apply(pub_rec_bankruptc
```

emp_length

In [53]:
```
train_df.drop('emp_length', axis=1, inplace=True)
test_df.drop('emp_length', axis=1, inplace=True)
```

5.2: Handle Missing Values

In [54]:
```
train_df.dtypes
```

Out[54]:
```
annual_inc               float64
earliest_cr_line           int64
grade                     object
home_ownership            object
application_type          object
initial_list_status       object
int_rate                 float64
loan_amnt                  int64
num_actv_bc_tl            object
mort_acc                 float64
tot_cur_bal              float64
open_acc                  object
pub_rec                    int64
pub_rec_bankruptcies     float64
revol_bal                  int64
revol_util               float64
term                       int64
total_acc                 object
verification_status       object
loan_status               object
region                    object
fico_score_category     category
purpose_category          object
dtype: object
```

In [55]:
```
train_df['open_acc'] = train_df['open_acc'].astype(float)
train_df['num_actv_bc_tl'] = train_df['num_actv_bc_tl'].astype(float)
train_df['total_acc'] = train_df['total_acc'].astype(float)

test_df['open_acc'] = test_df['open_acc'].astype(float)
test_df['num_actv_bc_tl'] = test_df['num_actv_bc_tl'].astype(float)
test_df['total_acc'] = test_df['total_acc'].astype(float)
```

In [ ]:
```

```

In [56]:
```
# Training dataset
missing_values_percentage = (train_df.isnull().sum() / len(train_df)) * 100
missing_values_percentage[missing_values_percentage > 0]
```

Out[56]:
```
num_actv_bc_tl          4.93500
mort_acc                3.46375
tot_cur_bal             4.93500
pub_rec_bankruptcies    0.03875
revol_util              0.06625
dtype: float64
```

In [57]:
```
# Test dataset
missing_values_percentage = (test_df.isnull().sum() / len(test_df)) * 100
missing_values_percentage[missing_values_percentage > 0]
```

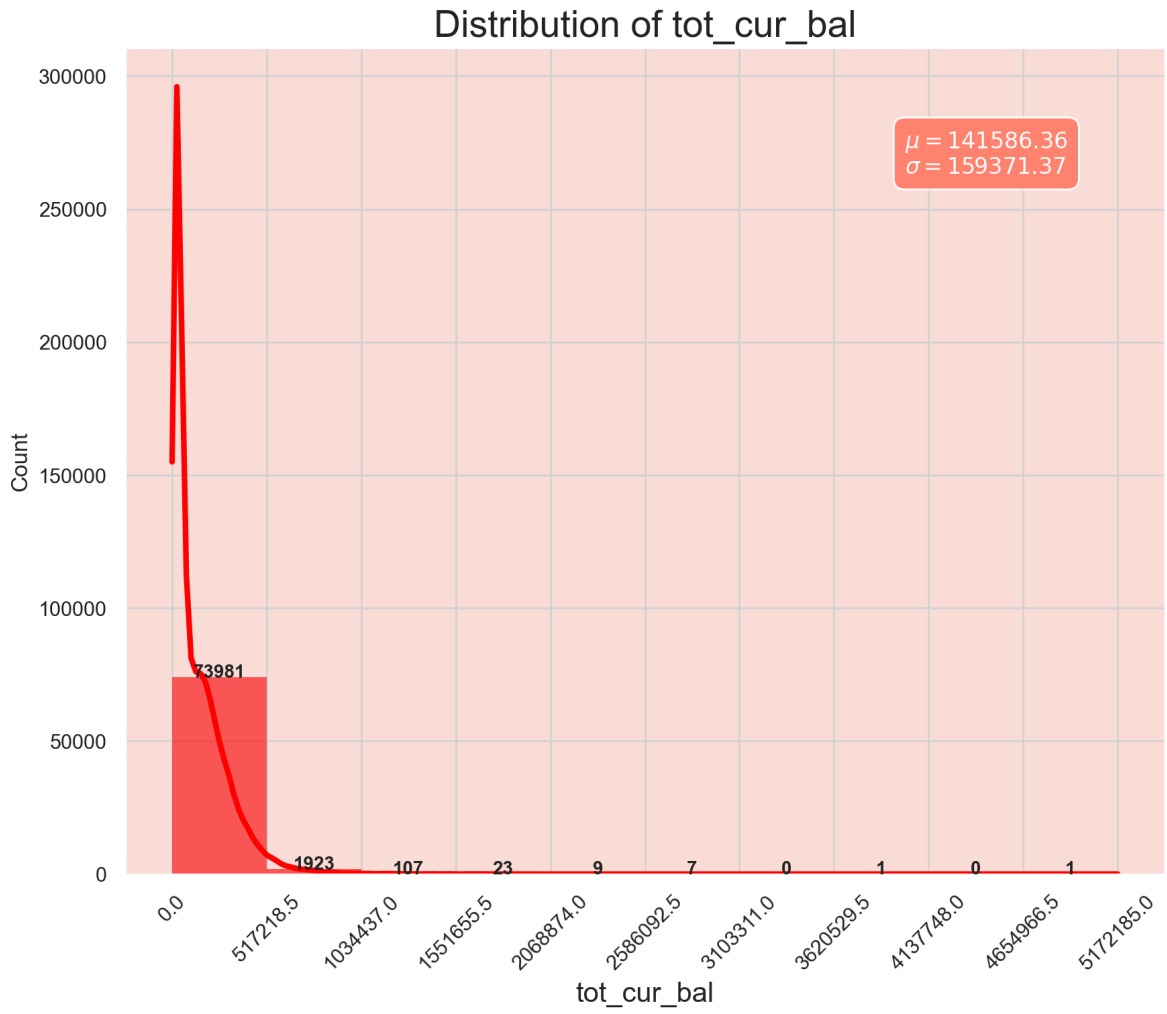
Out[57]:
```
num_actv_bc_tl          5.055
mort_acc                3.520
tot_cur_bal             5.055
pub_rec_bankruptcies    0.055
revol_util              0.065
dtype: float64
```

num_actv_bc_tl

In [58]:
```
train_df['num_actv_bc_tl'].fillna(0, inplace=True)
test_df['num_actv_bc_tl'].fillna(0, inplace=True)
```

tot_cur_bal

In [59]:
```
col = 'tot_cur_bal'

# Set up the plot
fig, ax = plt.subplots(figsize=(10, 8))

# Plot the histogram
values, bin_edges = np.histogram(train_df[col],
                                 range=(np.floor(train_df[col].min()), np.ceil(train_d
graph = sns.histplot(data=train_df, x=col, bins=bin_edges, kde=True, ax=ax,
                     edgecolor='none', color='red', alpha=0.6, line_kws={'lw': 3})
ax.set_xlabel(col, fontsize=15)
ax.set_ylabel('Count', fontsize=12)
ax.set_xticks(np.round(bin_edges, 1))
ax.set_xticklabels(ax.get_xticks(), rotation=45)
ax.grid(color='lightgrey')

# Annotate each bar with its count
for j, p in enumerate(graph.patches):
    ax.annotate('{}'.format(p.get_height()), (p.get_x() + p.get_width() / 2, p.get_hei
                ha='center', fontsize=10, fontweight="bold")

# Add text box with mean and standard deviation
textstr = '\n'.join((
    r'$\mu=%.2f$' % df_continuous[col].mean(),
    r'$\sigma=%.2f$' % df_continuous[col].std()
))
ax.text(0.75, 0.9, textstr, transform=ax.transAxes, fontsize=12, verticalalignment='to
        color='white', bbox=dict(boxstyle='round', facecolor='#ff826e', edgecolor='whi

# Set the title
plt.title(f'Distribution of {col}', fontsize=20)

# Show the plot
plt.show()
```



In [60]:
```
median_tot_cur_bal = train_df['tot_cur_bal'].median()
```

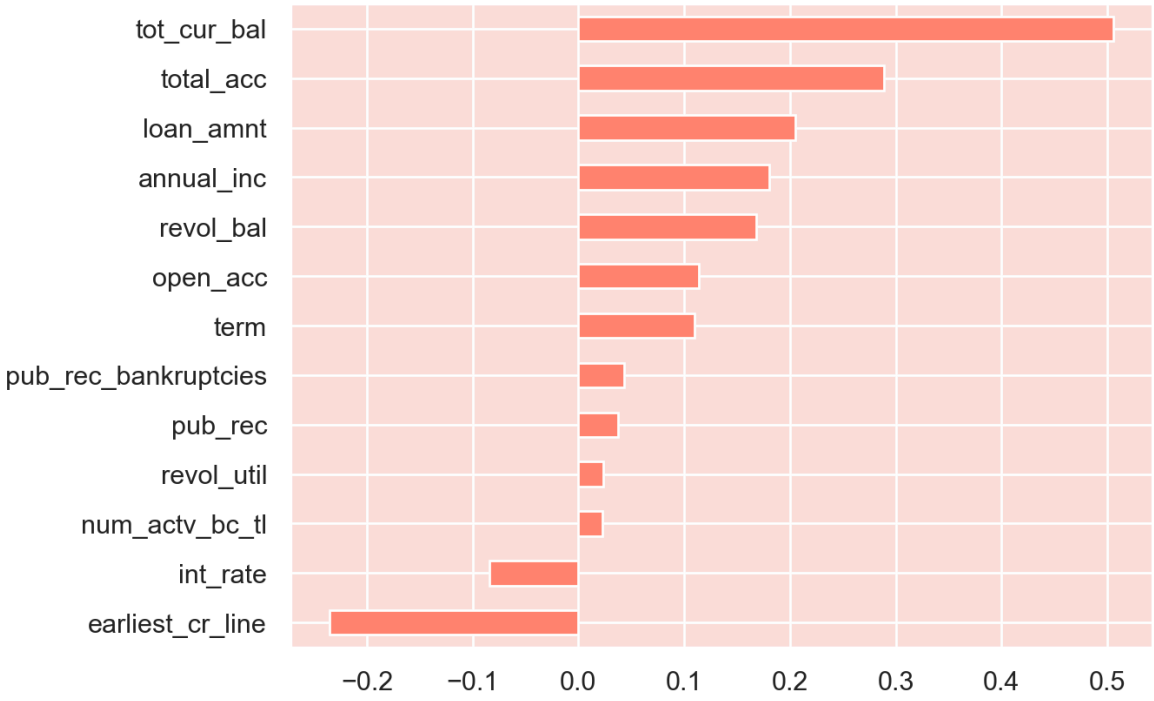
In [61]:
```
# Fill missing values with the calculated median
train_df['tot_cur_bal'].fillna(median_tot_cur_bal, inplace=True)
```

In [62]:
```
median_tot_cur_bal = test_df['tot_cur_bal'].median()
test_df['tot_cur_bal'].fillna(median_tot_cur_bal, inplace=True)
```

mort_acc

In [63]:
```
# we will check if this feature is correlated to any other feature in dataset
train_df.corr()['mort_acc'].drop('mort_acc').sort_values().plot.barh()
```

Out[63]: <Axes: >



Feature Engineering Action:

Observing the correlation between the **tot_cur_bal** and **mort_acc** features, it seems plausible that the number of mortgage accounts (**mort_acc**) may be related to the total current balance (**tot_cur_bal**). To address missing values in the **mort_acc** feature, we will employ the `fillna()` approach. This involves grouping the dataframe by **tot_cur_bal** and calculating the mean value for **mort_acc** per **tot_cur_bal** entry.

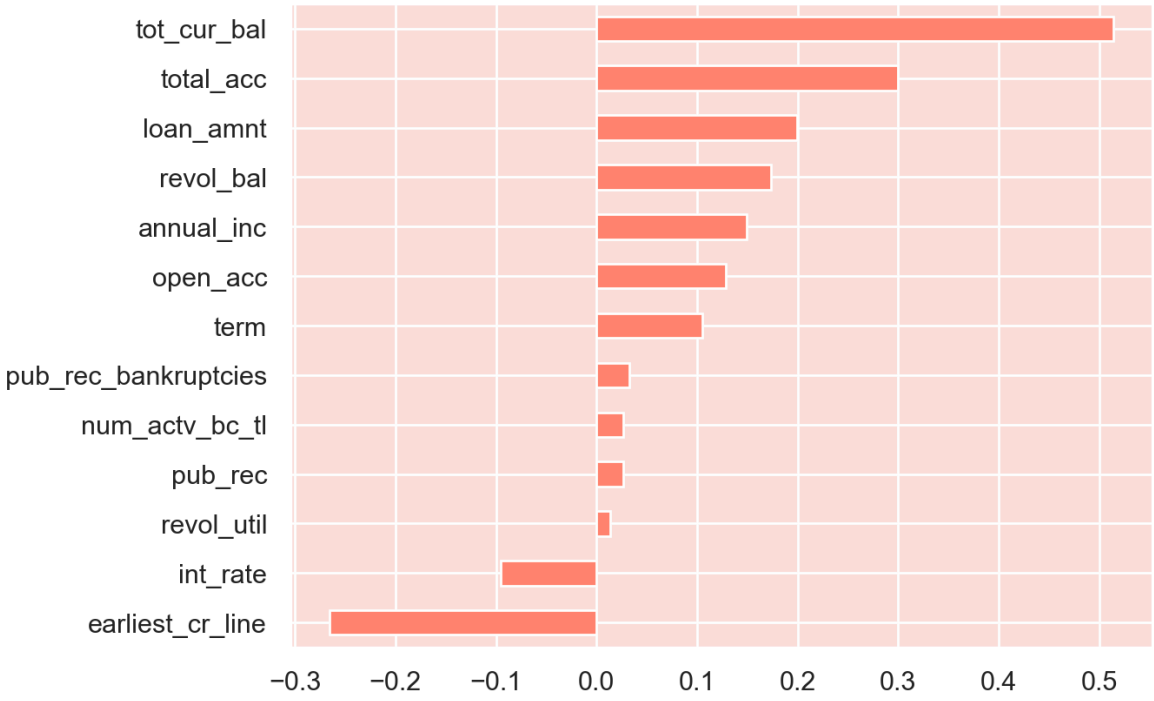
In [64]:
```
tot_cur_bal_avg = train_df.groupby(by='tot_cur_bal').mean().mort_acc
```

In [65]:
```
def fill_mort_acc(tot_cur_bal, mort_acc):
    if np.isnan(mort_acc):
        return tot_cur_bal_avg[tot_cur_bal].round()
    else:
        return mort_acc
```

In [66]:
```
train_df['mort_acc'] = train_df.apply(lambda x: fill_mort_acc(x['tot_cur_bal'], x['mor
```

In [67]:
```
## simialrly doing for test data
test_df.corr()['mort_acc'].drop('mort_acc').sort_values().plot.barh()
```

Out[67]: <Axes: >



In [68]:
```
tot_cur_bal_avg = test_df.groupby(by='tot_cur_bal').mean().mort_acc
test_df['mort_acc'] = test_df.apply(lambda x: fill_mort_acc(x['tot_cur_bal'], x['mort_
```

revol_util and pub_rec_bankruptcies

Inference:

These two features have missing data points, but they account for less than 0.5% of the total data. So we are going to remove the rows that are missing those values in those columns with dropna().

In [69]:
```
train_df.dropna(inplace=True)
test_df.dropna(inplace=True)
```

5.3: Outlier Treatment

⬆️ Table Contents

In [70]:
```
continuous_features
```

Out[70]:
```
Index(['annual_inc', 'int_rate', 'loan_amnt', 'tot_cur_bal', 'revol_bal',
       'revol_util'],
      dtype='object')
```

In [71]:
```
Q1 = train_df[continuous_features].quantile(0.25)
Q3 = train_df[continuous_features].quantile(0.75)
IQR = Q3 - Q1
outliers_count_specified = ((train_df[continuous_features] < (Q1 - 1.5 * IQR)) | (trai

outliers_count_specified
```

Out[71]:
```
annual_inc     3886
int_rate       1494
loan_amnt       432
tot_cur_bal    3046
revol_bal      4799
revol_util        3
dtype: int64
```

Outliers Identification:

Upon identifying outliers for the specified continuous features, we found the following counts of outliers:

- **annual_inc:** 3886 outliers
- **int_rate:** 1494 outliers
- **loan_amnt:** 432 outliers
- **tot_cur_bal:** 3046 outliers
- **revol_bal:** 4799 outliers
- **revol_util:** 3 outliers

In [72]:
```
Q1 = train_df[continuous_features].quantile(0.25)
Q3 = train_df[continuous_features].quantile(0.75)
IQR = Q3 - Q1
train_outliers_count_specified = (((train_df[continuous_features] < (Q1 - 1.5 * IQR))

print("Outlier Ratio's Train Data:\n",train_outliers_count_specified)


Q1 = test_df[continuous_features].quantile(0.25)
Q3 = test_df[continuous_features].quantile(0.75)
IQR = Q3 - Q1
test_outliers_count_specified = (((test_df[continuous_features] < (Q1 - 1.5 * IQR)) |

print("\nOutlier Ratio's Test Data:\n",test_outliers_count_specified)
```

```
Outlier Ratio's Train Data:
 annual_inc     0.048626
int_rate       0.018695
loan_amnt      0.005406
tot_cur_bal    0.038115
revol_bal      0.060051
revol_util     0.000038
dtype: float64

Outlier Ratio's Test Data:
 annual_inc     0.048708
int_rate       0.017871
loan_amnt      0.005356
tot_cur_bal    0.037795
revol_bal      0.056818
revol_util     0.000050
dtype: float64
```

Outlier Ratios Comparison:

Comparing the outlier ratios between the train and test datasets, we can see that they are generally similar for most features. This suggests that the outlier counts are balanced relative to the number of rows in both datasets.

📝 Sensitivity to Outliers:

For model evaluations, we are going to use Tree Based Models:
Decision Trees (DT) and Random Forests (RF), these tree-based algorithms are generally robust to outliers. They make splits based on feature values, and outliers often end up in leaf nodes, having minimal impact on the overall decision-making process.
AdaBoost: This ensemble method, which often uses decision trees as weak learners, is generally robust to outliers. However, the iterative nature of AdaBoost can sometimes lead to overemphasis on outliers, making the final model more sensitive to them.

To overcome this we'll focus on applying transformations like **Box-Cox** in the subsequent steps to reduce the impact of outliers and make the data more suitable for modeling.

5.4: Duplicate Data

⬆️ Table Contents

In [73]:
```
train_df.drop_duplicates(inplace=True)
test_df.drop_duplicates(inplace=True)
```

5.5: Encode Categorical Variables

⬆️ Table Contents

Categorization of Features for Encoding:

After analyzing the dataset, we can categorize the features into three groups:

1. **No Encoding Needed:** These are the features that do not require any form of encoding because they are already in a numerical format that can be fed into a model.
2. **One-Hot Encoding:** This is required for nominal variables, which are categorical variables without any intrinsic order. One-hot encoding converts each unique value of the feature into a separate column with a 1 or 0, indicating the presence of that value.
3. **Label Encoding:** This is used for ordinal variables, which are categorical variables with a meaningful order. Label encoding assigns a unique integer to each category in the feature, maintaining the order of the values.

By categorizing the features into these groups, we can apply the appropriate encoding method to each feature, preparing the dataset for modeling.

In [ ]:
```
```

In [74]:
```
train_df.dtypes
```

Out[74]:
```
annual_inc               float64
earliest_cr_line           int64
grade                     object
home_ownership            object
application_type          object
initial_list_status       object
int_rate                 float64
loan_amnt                  int64
num_actv_bc_tl           float64
mort_acc                 float64
tot_cur_bal              float64
open_acc                 float64
pub_rec                    int64
pub_rec_bankruptcies     float64
revol_bal                  int64
revol_util               float64
term                       int64
total_acc                float64
verification_status       object
loan_status               object
region                    object
fico_score_category     category
purpose_category          object
dtype: object
```

No Encoding Needed:

- annual_inc
- earliest_cr_line
- int_rate
- loan_amnt
- num_actv_bc_tl
- mort_acc
- tot_cur_bal
- open_acc
- pub_rec
- pub_rec_bankruptcies
- revol_bal
- revol_util
- term
- total_acc

One-Hot Encoding:

- home_ownership
- application_type
- initial_list_status
- verification_status
- region
- purpose_category

Label Encoding:

- loan_status
- grade
- fico_score_category

One-Hot-Encoding

In [75]:
```
one_hot_cols = ['home_ownership','application_type','initial_list_status','verificatio
```

In [76]:
```
train_df = pd.get_dummies(train_df, columns=one_hot_cols, drop_first=True)
test_df = pd.get_dummies(test_df, columns=one_hot_cols, drop_first=True)
```

In [77]:
```
train_df['grade'].unique()
```

Out[77]:
```
array(['E', 'B', 'F', 'D', 'C', 'A', 'G'], dtype=object)
```

Label-Encoding

In [78]:
```
# Define the encoding dictionary
grade_encoding = {'A': 7, 'B': 6, 'C': 5, 'D': 4, 'E': 3, 'F': 2, 'G': 1}

# Map the encoding to the 'grade' column
train_df['grade_encoded'] = train_df['grade'].map(grade_encoding)
train_df.drop(columns=['grade'], inplace=True)

test_df['grade_encoded'] = test_df['grade'].map(grade_encoding)
test_df.drop(columns=['grade'], inplace=True)
```

In [79]:
```
# Define the encoding dictionary
fico_encoding = {'Exceptional': 5, 'Very Good': 4, 'Good': 3, 'Fair': 2, 'Poor': 1}

# Map the encoding to the 'grade' column
train_df['fico_score_category_enc'] = train_df['fico_score_category'].map(fico_encodin
train_df.drop(columns=['fico_score_category'], inplace=True)

test_df['fico_score_category_enc'] = test_df['fico_score_category'].map(fico_encoding)
test_df.drop(columns=['fico_score_category'], inplace=True)
```

In [80]:
```
train_df['fico_score_category_enc'] = train_df['fico_score_category_enc'].astype(int)
test_df['fico_score_category_enc'] = test_df['fico_score_category_enc'].astype(int)
```

Loan Status Encoding | 1: Defaulted, 0: Paid

In [81]:
```
# Target Feature encoding
target_encoded = {'Paid': 0, 'Defaulted': 1}
train_df['loan_status'] = train_df['loan_status'].map(target_encoded)
```

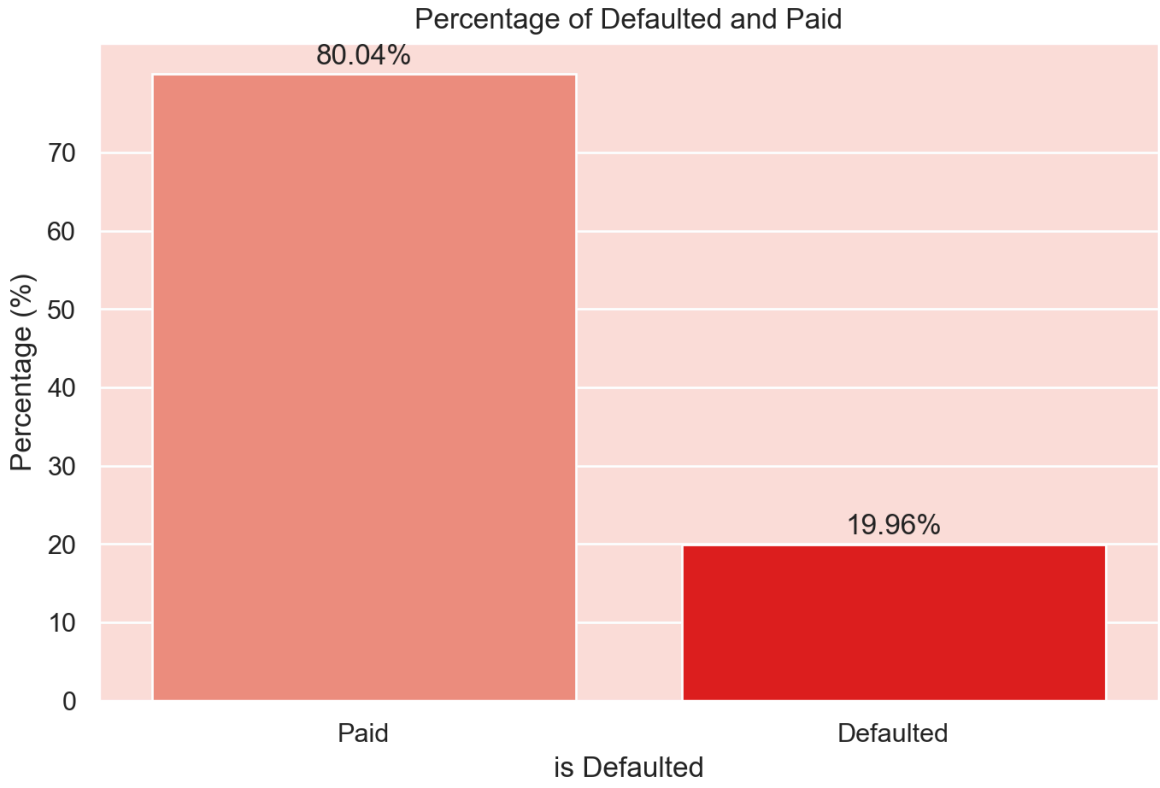
In [82]:
```
train_df['loan_status'].value_counts()
```

Out[82]:
```
0    63963
1    15953
Name: loan_status, dtype: int64
```

5.6: Check Imbalanced Data

⬆️ Table Contents

In [83]:
```
# Calculating the percentage of each class
percentage = train_df['loan_status'].value_counts(normalize=True) * 100

# Plotting the percentage of each class
plt.figure(figsize=(8, 5))
ax = sns.barplot(x=percentage.index, y=percentage, palette=['#ff826e', 'red'])
plt.title('Percentage of Defaulted and Paid')
plt.xlabel('is Defaulted')
plt.ylabel('Percentage (%)')
plt.xticks(ticks=[0, 1], labels=['Paid','Defaulted'])
plt.yticks(ticks=range(0,80,10))

# Displaying the percentage on the bars
for i, p in enumerate(percentage):
    ax.text(i, p + 0.5, f'{p:.2f}%', ha='center', va='bottom')

plt.show()
```



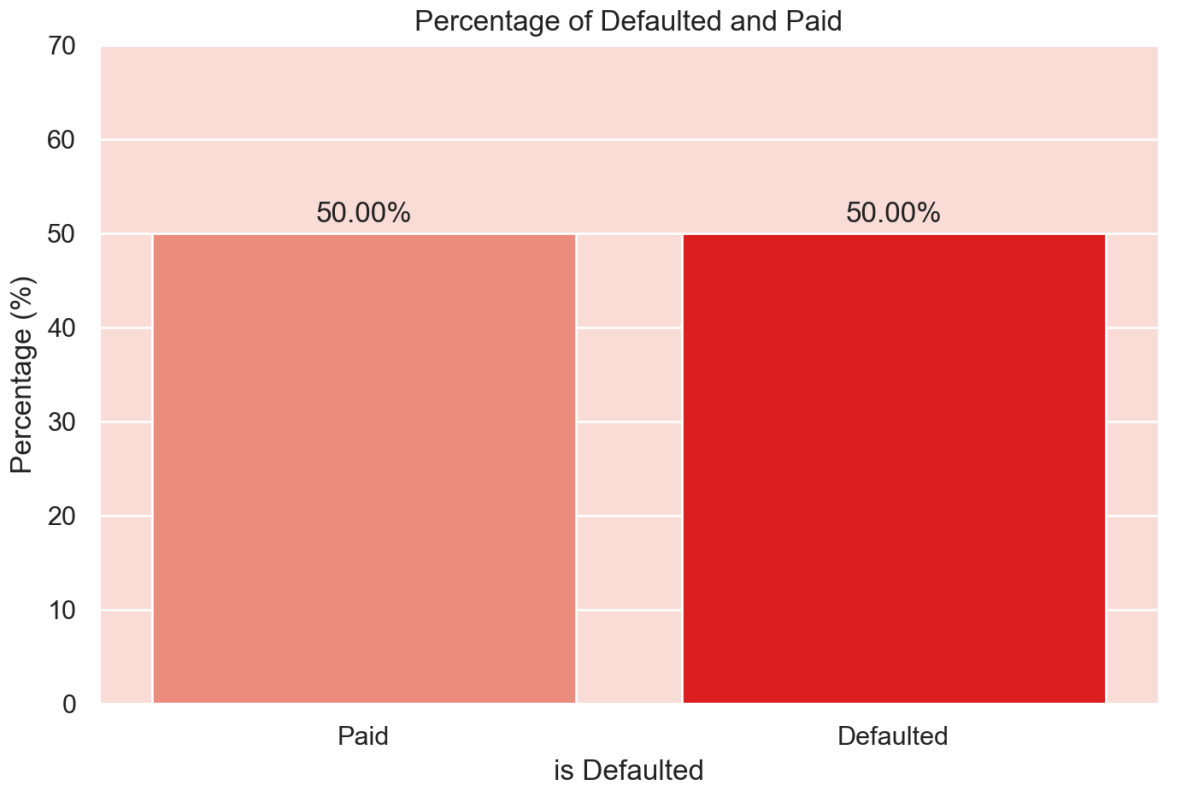
The bar plot shows the percentage of defaulted and paid in the dataset. Approximately 80.04% of the loan status was paid, and 19.96% were defaulted. This indicates that there is high imbalance in the target variable. To address this, we will use **SMOTE (Synthetic Minority Over-sampling Technique)**. SMOTE is a technique used to generate synthetic samples for the minority class in order to balance the class distribution in the dataset. By creating synthetic samples, SMOTE helps mitigate the impact of class imbalance and improves the performance of machine learning models in predicting the minority class.

5.6.1: Handling Unbalanced

⬆️ Table Contents

In [84]:
```
from imblearn.over_sampling import SMOTE

sm = SMOTE(sampling_strategy='minority', random_state=42)
# Fit the model to generate the data.
oversampled_X, oversampled_Y = sm.fit_resample(train_df.drop('loan_status', axis=1), t
oversampled = pd.concat([pd.DataFrame(oversampled_Y), pd.DataFrame(oversampled_X)], ax
```

In [85]:
```
# Calculating the percentage of each class
percentage = oversampled['loan_status'].value_counts(normalize=True) * 100

# Plotting the percentage of each class
plt.figure(figsize=(8, 5))
ax = sns.barplot(x=percentage.index, y=percentage, palette=['#ff826e', 'red'])
plt.title('Percentage of Defaulted and Paid')
plt.xlabel('is Defaulted')
plt.ylabel('Percentage (%)')
plt.xticks(ticks=[0, 1], labels=['Paid','Defaulted'])
plt.yticks(ticks=range(0,80,10))

# Displaying the percentage on the bars
for i, p in enumerate(percentage):
    ax.text(i, p + 0.5, f'{p:.2f}%', ha='center', va='bottom')

plt.show()
```



6: Business Requirement

⬆️ Table Contents

> 📝 **Note:**
> In the context of loan default prediction:
>
> - **Prioritize High Recall (Sensitivity) for Defaulted Loans**: Emphasize identifying most of the actual defaulted loans correctly, even if it leads to some false positives (paid loans being

misclassified as defaulted). It's crucial to capture as many true defaulted cases as possible to mitigate financial risks and take necessary actions.

- **Minimize False Negatives (FN)**: Aim to reduce instances where defaulted loans are missed by the model. Missing defaulted cases could result in financial losses and impact the overall portfolio performance.
- **Balance Precision and Recall (F1-score)**: While minimizing false positives is important to avoid unnecessary interventions or restrictions on credit access, prioritize achieving high recall to ensure defaulted loans are not overlooked by the model.**the F1-score for the 'defaulted' class (1) would be the most important metric for evaluating models in this project**.

By focusing on these aspects, the loan default prediction model can effectively identify loans at risk of default, enabling proactive measures to mitigate potential losses and maintain a healthy loan portfolio.

7: Spliting Training Dataset

⬆️ Table Contents

In [86]:
```
X = oversampled.drop(['loan_status'], axis=1)
y = oversampled['loan_status']
```

In [87]:
```
# Split the labeled training data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42
```

8: Decision Tree Model Building

⬆️ Table Contents

8.1: DT Base Model Definition

⬆️ Table Contents

In [88]:
```
# Define the base DT model
dt_base = DecisionTreeClassifier(random_state=0)
```

8.2: DT Hyperparameter Tuning

⬆️ Table Contents

I will create a function to identify the best set of hyperparameters that maximize the F1-score for class 1 (defaulted). This method provides a reusable framework for **hyperparameter tuning** for other models as well,It uses **GridSearchCV and cross-validation (StratifiedKFold)** to evaluate different combinations of hyperparameters.

In [89]:
```
def tune_clf_hyperparameters(clf, param_grid, X_train, y_train, scoring='f1', n_splits
    '''
    This function optimizes the hyperparameters for a classifier by searching over a s
    It uses GridSearchCV and cross-validation (StratifiedKFold) to evaluate different
    The combination with the highest F1-score for class 1 (defaulted) is selected as t
    The function returns the classifier with the optimal hyperparameters.
    '''

    # Create the cross-validation object using StratifiedKFold to ensure the class dis
    cv = StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=0)

    # Create the GridSearchCV object
    clf_grid = GridSearchCV(clf, param_grid, cv=cv, scoring=scoring, n_jobs=-1)

    # Fit the GridSearchCV object to the training data
    clf_grid.fit(X_train, y_train)

    # Get the best hyperparameters
    best_hyperparameters = clf_grid.best_params_

    # Return best_estimator_ attribute which gives us the best model that has been fit
    return clf_grid.best_estimator_, best_hyperparameters
```

In [90]:
```
# Hyperparameter grid for DT
param_grid_dt = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [23, 24, 25, 26, 27],
    'min_samples_split': [2, 3, 4, 5],
    'min_samples_leaf': [1, 2, 3, 4],


    # we want to optimize for class 1, we have included the class_weight parameter in
    # In the grid above, the weight for class 0 is always 1, while the weight for clas
    # This will help the model to focus more on class 1.
}
```

In [91]:
```
# Call the function for hyperparameter tuning
best_dt, best_dt_hyperparams = tune_clf_hyperparameters(dt_base, param_grid_dt, X_trai
```

In [92]:
```
print('DT Optimal Hyperparameters: \n', best_dt_hyperparams)
```

```
DT Optimal Hyperparameters:
 {'criterion': 'entropy', 'max_depth': 23, 'min_samples_leaf': 1, 'min_samples_spli
t': 2}
```

8.3: DT Model Evaluation

⬆️ Table Contents

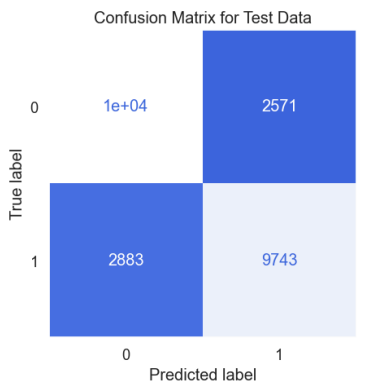
To streamline the evaluation of different models, we will define a set of functions that compute key performance metrics. This approach will ensure consistency in how we assess each model and facilitate comparisons between them

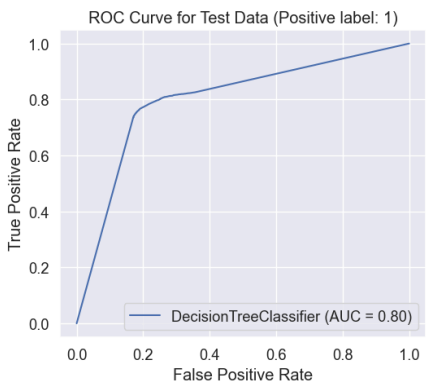
In [93]:
```
def metrics_calculator(clf, X_test, y_test, model_name):
    '''
    This function calculates all desired performance metrics for a given model on test
    The metrics are calculated specifically for class 1.
    '''
    y_pred = clf.predict(X_test)
    result = pd.DataFrame(data=[accuracy_score(y_test, y_pred),
                                precision_score(y_test, y_pred, pos_label=1),
                                recall_score(y_test, y_pred, pos_label=1),
                                f1_score(y_test, y_pred, pos_label=1),
                                roc_auc_score(y_test, clf.predict_proba(X_test)[:,1])]
                          index=['Accuracy','Precision (Class 1)','Recall (Class 1)','
                          columns = [model_name])

    result = (result * 100).round(2).astype(str) + '%'
    return result
```

In [94]:
```
def model_evaluation(clf, X_train, X_test, y_train, y_test, model_name):
    '''
    This function provides a complete report of the model's performance including clas
    confusion matrix and ROC curve.
    '''
    sns.set(font_scale=1.2)

    # Generate classification report for training set
    y_pred_train = clf.predict(X_train)
    print("\n\t  Classification report for training set")
    print("-"*55)
    print(classification_report(y_train, y_pred_train))

    # Generate classification report for test set
    y_pred_test = clf.predict(X_test)
    print("\n\t   Classification report for validation test set")
    print("-"*55)
    print(classification_report(y_test, y_pred_test))

    # Create figure and subplots
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 5), dpi=100, gridspec_kw={'

    # Define a colormap
    royalblue = LinearSegmentedColormap.from_list('royalblue', [(0, (1,1,1)), (1, (0.2
    royalblue_r = royalblue.reversed()

    # Plot confusion matrix for test set
    ConfusionMatrixDisplay.from_estimator(clf, X_test, y_test, colorbar=False, cmap=ro
    ax1.set_title('Confusion Matrix for Test Data')
    ax1.grid(False)

    # Plot ROC curve for test data and display AUC score
    RocCurveDisplay.from_estimator(clf, X_test, y_test, ax=ax2)
    ax2.set_xlabel('False Positive Rate')
    ax2.set_ylabel('True Positive Rate')
    ax2.set_title('ROC Curve for Test Data (Positive label: 1)')

    # Report results for the class specified by positive label
    result = metrics_calculator(clf, X_test, y_test, model_name)
    table = ax3.table(cellText=result.values, colLabels=result.columns, rowLabels=resu
    table.scale(0.6, 2)
    table.set_fontsize(12)
    ax3.axis('tight')
    ax3.axis('off')
    # Modify color
    for key, cell in table.get_celld().items():
        if key[0] == 0:
            cell.set_color('royalblue')
    plt.tight_layout()
    plt.show()
```

In [95]:
```
model_evaluation(best_dt, X_train, X_val, y_train, y_val, 'Decision Tree')
```

```
          Classification report for training set
-----------------------------------------------------------
              precision    recall  f1-score   support

           0       0.95      0.98      0.97     51003
           1       0.98      0.95      0.97     51337

    accuracy                           0.97    102340
   macro avg       0.97      0.97      0.97    102340
weighted avg       0.97      0.97      0.97    102340


          Classification report for validation test set
-----------------------------------------------------------
              precision    recall  f1-score   support

           0       0.78      0.80      0.79     12960
           1       0.79      0.77      0.78     12626

    accuracy                           0.79     25586
   macro avg       0.79      0.79      0.79     25586
weighted avg       0.79      0.79      0.79     25586
```





	Decision Tree
Accuracy	78.68%
Precision (Class 1)	79.12%
Recall (Class 1)	77.17%
F1-score (Class 1)	78.13%
AUC (Class 1)	79.9%

In [96]:
```
# Save the final performance of DT classifier
dt_result = metrics_calculator(best_dt, X_val, y_val, 'Decision Tree')
dt_result
```

Out[96]:

	Decision Tree
Accuracy	78.68%
Precision (Class 1)	79.12%
Recall (Class 1)	77.17%
F1-score (Class 1)	78.13%
AUC (Class 1)	79.9%

Decision Tree Model Evaluation (Loan Domain)

Metric	Value	Interpretation
Accuracy	78.68%	The model correctly predicted loan statuses for 78.68% of the cases.
Precision (Defaulted)	79.12%	Out of all loans predicted as defaulted, only 79.12% were actually defaulted.
Recall (Defaulted)	77.17%	The model identified 38% of the actual defaulted loans.

F1-score (Defaulted)	78.13%	The harmonic mean of precision and recall for defaulted loans is 78.13%.
AUC (Defaulted)	79.9%	The Area Under the ROC Curve (AUC) for defaulted loans is 80%, indicating the model's ability to rank defaulted loans higher than non-defaulted ones.

The evaluation of the Decision Tree model in the loan domain reveals its performance in predicting defaulted loans. While it achieved an accuracy of 78.68%, indicating overall correctness, the precision, recall, and F1-score for defaulted loans suggest room for improvement in identifying actual defaulted cases. The AUC score further validates the model's discriminative power in distinguishing defaulted loans from paid ones.

8.4: Target Prediction for Test Data

⬆️ Table Contents

In [97]:
```
y_test_pred = best_dt.predict(test_df)
```

In [98]:
```
# Create a DataFrame from y_test_pred with column name "predicted_loan_status"
predictions_df = pd.DataFrame(data=y_test_pred, columns=["predicted_loan_status"])

DT_result_df = pd.concat([test_df.reset_index(drop=True), predictions_df], axis=1)
```

In [99]:
```
DT_result_df.head()
```

Out[99]:

	annual_inc	earliest_cr_line	int_rate	loan_amnt	num_actv_bc_tl	mort_acc	tot_cur_bal	open_acc	pub
0	50000.0	2012	13.99	5000.0	1.0	0.0	33395.0	9.0	
1	92000.0	2001	10.99	30000.0	2.0	1.0	229832.0	11.0	
2	89000.0	1989	10.15	16000.0	5.0	1.0	181616.0	15.0	
3	33000.0	2004	13.68	10000.0	6.0	0.0	30603.0	12.0	
4	35580.0	1997	14.09	4000.0	2.0	1.0	124597.0	8.0	

5 rows × 31 columns

9: Random Forest Model Building

⬆️ Table Contents

9.1: RF Base Model Definition

⬆️ Table Contents

In [104]:
```
# Define the base RF model
rf_base = RandomForestClassifier(random_state=0, n_jobs=-1)
```

9.2: RF Hyperparameter Tuning

⬆️ Table Contents

In [105]:
```
param_grid_rf = {
    'n_estimators': [100, 150],
    'criterion': ['entropy'],
    'max_depth': [16, 18],
    'min_samples_split': [2, 3, 4],
    'min_samples_leaf': [1, 2, 3],
}
```

In [106]:
```
# Using the tune_clf_hyperparameters function to get the best estimator
best_rf, best_rf_hyperparams = tune_clf_hyperparameters(rf_base, param_grid_rf, X_trai
```

In [107]:
```
print('RF Optimal Hyperparameters: \n', best_rf_hyperparams)
```

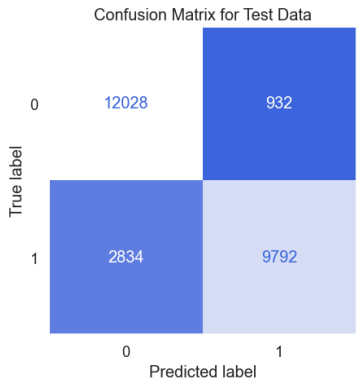
```
RF Optimal Hyperparameters:
 {'criterion': 'entropy', 'max_depth': 18, 'min_samples_leaf': 1, 'min_samples_spli
t': 2, 'n_estimators': 150}
```

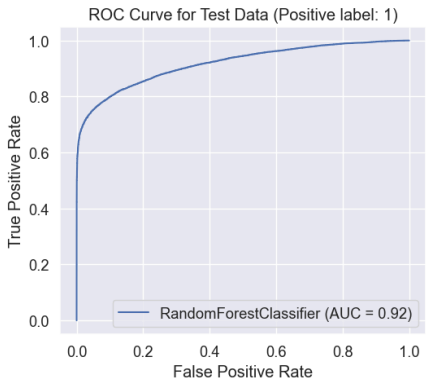
9.3: RF Model Evaluation

⬆️ Table Contents

In [108]:
```
model_evaluation(best_rf, X_train, X_val, y_train, y_val, 'Random Forest')
```

```
          Classification report for training set
-----------------------------------------------------------
              precision    recall  f1-score   support

           0       0.92      0.98      0.95     51003
           1       0.98      0.92      0.95     51337

    accuracy                           0.95    102340
   macro avg       0.95      0.95      0.95    102340
weighted avg       0.95      0.95      0.95    102340


          Classification report for validation test set
-----------------------------------------------------------
              precision    recall  f1-score   support

           0       0.81      0.93      0.86     12960
           1       0.91      0.78      0.84     12626

    accuracy                           0.85     25586
   macro avg       0.86      0.85      0.85     25586
weighted avg       0.86      0.85      0.85     25586
```





	Random Forest
Accuracy	85.28%
Precision (Class 1)	91.31%
Recall (Class 1)	77.55%
F1-score (Class 1)	83.87%
AUC (Class 1)	91.82%

RF Model Evaluation (Loan Domain)

Metric	Value	Interpretation
Accuracy	85.28%	The model correctly predicted loan statuses for 85% of the cases.
Precision (Defaulted)	91.31%	Out of all loans predicted as defaulted, only 91% were actually defaulted.
Recall (Defaulted)	77.55%	The model identified 76% of the actual defaulted loans.
F1-score (Defaulted)	83.87%	The harmonic mean of precision and recall for defaulted loans is 84%.
AUC (Defaulted)	91.82%	The Area Under the ROC Curve (AUC) for defaulted loans is 92%, indicating the model's ability to rank defaulted loans higher than non-defaulted ones.

The evaluation of the RF model in the loan domain reveals its performance in predicting defaulted loans. While it achieved an accuracy of 85%, indicating overall correctness, the precision, recall, and F1-score for defaulted loans suggest room for improvement in identifying actual defaulted cases. The AUC score further validates the model's discriminative power in distinguishing defaulted loans from paid ones.

In [109]:
```
# Save the final performance of RF classifier
rf_result = metrics_calculator(best_rf, X_val, y_val, 'Random Forest')
rf_result
```

Out[109]:

	Random Forest
Accuracy	85.28%
Precision (Class 1)	91.31%
Recall (Class 1)	77.55%
F1-score (Class 1)	83.87%
AUC (Class 1)	91.82%

9.4: Target Prediction for Test Data

⬆️ Table Contents

In [110]:
```
y_test_pred = best_rf.predict(test_df)
```

In [111]:
```
# Create a DataFrame from y_test_pred with column name "predicted_loan_status"
predictions_df = pd.DataFrame(data=y_test_pred, columns=["predicted_loan_status"])

RF_result_df = pd.concat([test_df.reset_index(drop=True), predictions_df], axis=1)
```

In [112]:
```
RF_result_df.head()
```

Out[112]:

	annual_inc	earliest_cr_line	int_rate	loan_amnt	num_actv_bc_tl	mort_acc	tot_cur_bal	open_acc	pub
0	50000.0	2012	13.99	5000.0	1.0	0.0	33395.0	9.0	
1	92000.0	2001	10.99	30000.0	2.0	1.0	229832.0	11.0	
2	89000.0	1989	10.15	16000.0	5.0	1.0	181616.0	15.0	
3	33000.0	2004	13.68	10000.0	6.0	0.0	30603.0	12.0	
4	35580.0	1997	14.09	4000.0	2.0	1.0	124597.0	8.0	

5 rows × 31 columns

10: XGBoost Model Building

⬆️ Table Contents

10.1: XGBoost Base Model Definition

In [113]:
```
# Define the model
xgb_base = xgb.XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_st
```

10:2: Outlier treating using Box-cox

⬆️ Table Contents

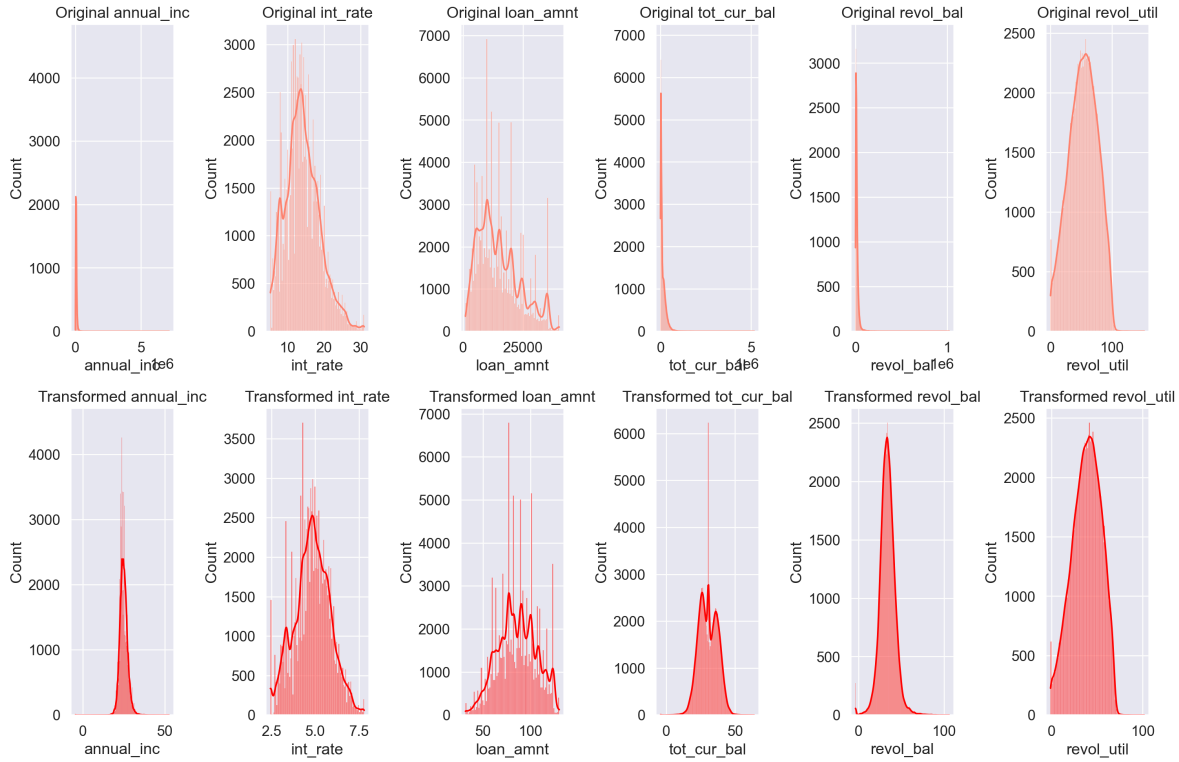
In [114]:
```
from scipy.stats import boxcox
```

In [115]:
```
continuous_features
```

Out[115]:
```
Index(['annual_inc', 'int_rate', 'loan_amnt', 'tot_cur_bal', 'revol_bal',
       'revol_util'],
      dtype='object')
```

In [116]:
```
# Adding a small constant to 'oldpeak' to make all values positive
X_train['annual_inc'] = X_train['annual_inc'] + 0.001
X_val['annual_inc'] = X_val['annual_inc'] + 0.001
test_df['annual_inc'] = test_df['annual_inc'] + 0.001

X_train['tot_cur_bal'] = X_train['tot_cur_bal'] + 0.001
X_val['tot_cur_bal'] = X_val['tot_cur_bal'] + 0.001
test_df['tot_cur_bal'] = test_df['tot_cur_bal'] + 0.001

X_train['revol_bal'] = X_train['revol_bal'] + 0.001
X_val['revol_bal'] = X_val['revol_bal'] + 0.001
test_df['revol_bal'] = test_df['revol_bal'] + 0.001

X_train['revol_util'] = X_train['revol_util'] + 0.001
X_val['revol_util'] = X_val['revol_util'] + 0.001
test_df['revol_util'] = test_df['revol_util'] + 0.001
```

In [117]:
```
fig, ax = plt.subplots(2, 6, figsize=(15,10))

# Original Distributions
for i, col in enumerate(continuous_features):
    sns.histplot(X_train[col], kde=True, ax=ax[0,i], color='#ff826e').set_title(f'Orig


# Applying Box-Cox Transformation
# Dictionary to store lambda values for each feature
lambdas = {}

for i, col in enumerate(continuous_features):
    # Only apply box-cox for positive values
    if X_train[col].min() > 0:
        X_train[col], lambdas[col] = boxcox(X_train[col])
        # Applying the same lambda to test data
        test_df[col] = boxcox(test_df[col], lmbda=lambdas[col])
        X_val[col] = boxcox(X_val[col], lmbda=lambdas[col])
        sns.histplot(X_train[col], kde=True, ax=ax[1,i], color='red').set_title(f'Tran
    else:
        sns.histplot(X_train[col], kde=True, ax=ax[1,i], color='green').set_title(f'{c

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



10.3: XGBoost Hyperparameter Tuning

⬆️ Table Contents

Due to the large number of XGBoost hyperparameters, the process of finding optimal hyperparameters will be very time-consuming. For this reason, we made the set of grid values smaller and finally we got the following optimal combination of hyperparameters for our XGBoost classifier:

In [118]:
```
# Initialize the XGBoost Classifier using optimal hyperparameters
xgb_opt = XGBClassifier(max_depth=6,
                        learning_rate=0.05,
                        n_estimators=200,
                        min_child_weight=2,
                        scale_pos_weight=0.5,
                        subsample=0.9 ,
                        colsample_bytree=0.5,
                        colsample_bylevel=0.8 ,
                        reg_alpha=0.05 ,
                        reg_lambda=0.1 ,
                        max_delta_step=2 ,
                        gamma=0.1,
                        random_state=0)

# Train the XGBoost classifier
xgb_opt.fit(X_train, y_train)
```

Out[118]:
```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=0.8, colsample_bynode=None,
              colsample_bytree=0.5, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=0.1, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.05, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None, max_delta_step=2,
              max_depth=6, max_leaves=None, min_child_weight=2, missing=nan,
              monotone_constraints=None, multi_strategy=None, n_estimators=200,
              n_jobs=None, num_parallel_tree=None, random_state=0, ...)
```

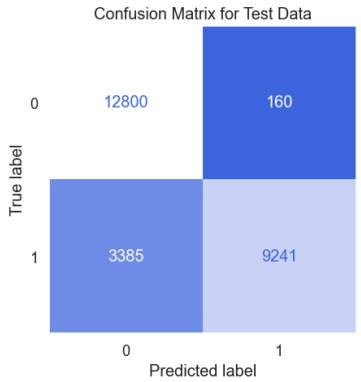
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

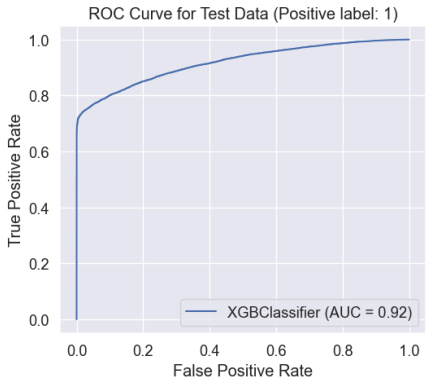
10.4: XGBoost Model Evaluation

⬆️ Table Contents

In [120]:
```
model_evaluation(xgb_opt, X_train, X_val, y_train, y_val, 'XGBoost')
```

```
          Classification report for training set
-----------------------------------------------------------
              precision    recall  f1-score   support

           0       0.79      0.99      0.88     51003
           1       0.99      0.74      0.85     51337

    accuracy                           0.86    102340
   macro avg       0.89      0.86      0.86    102340
weighted avg       0.89      0.86      0.86    102340


          Classification report for validation test set
-----------------------------------------------------------
              precision    recall  f1-score   support

           0       0.79      0.99      0.88     12960
           1       0.98      0.73      0.84     12626

    accuracy                           0.86     25586
   macro avg       0.89      0.86      0.86     25586
weighted avg       0.89      0.86      0.86     25586
```





	XGBoost
Accuracy	86.14%
Precision (Class 1)	98.3%
Recall (Class 1)	73.19%
F1-score (Class 1)	83.91%
AUC (Class 1)	91.8%

In [122]:
```
# Save the final performance of XGBoost classifier
xgb_result = metrics_calculator(xgb_opt, X_val, y_val, 'XGBoost')
xgb_result
```

Out[122]:

	XGBoost
Accuracy	86.14%
Precision (Class 1)	98.3%
Recall (Class 1)	73.19%
F1-score (Class 1)	83.91%
AUC (Class 1)	91.8%

XGBoost Model Evaluation (Loan Domain)

Metric	Value	Interpretation
Accuracy	86.14%	The model correctly predicted loan statuses for 86% of the cases.
Precision (Defaulted)	98.3%	Out of all loans predicted as defaulted, only 98% were actually defaulted.
Recall (Defaulted)	73.19%	The model identified 73% of the actual defaulted loans.

F1-score (Defaulted)	83.91%	The harmonic mean of precision and recall for defaulted loans is 84%.
AUC (Defaulted)	91.8%	The Area Under the ROC Curve (AUC) for defaulted loans is 92%, indicating the model's ability to rank defaulted loans higher than non-defaulted ones.

The evaluation of the XGBoost model in the loan domain reveals its performance in predicting defaulted loans. While it achieved an accuracy of 86%, indicating overall correctness, the precision, recall, and F1-score for defaulted loans suggest room for improvement in identifying actual defaulted cases,however compared to other models, XGBoost can be good model as it has 92% of Recall. The AUC score further validates the model's discriminative power in distinguishing defaulted loans from paid ones.

10.5: Target Prediction for Test Data

⬆️ Table Contents

In [124]:
```
y_test_pred = xgb_opt.predict(test_df)
```

In [125]:
```
# Create a DataFrame from y_test_pred with column name "predicted_loan_status"
predictions_df = pd.DataFrame(data=y_test_pred, columns=["predicted_loan_status"])

XG_result_df = pd.concat([test_df.reset_index(drop=True), predictions_df], axis=1)
```

In [126]:
```
XG_result_df.head()
```

Out[126]:

	annual_inc	earliest_cr_line	int_rate	loan_amnt	num_actv_bc_tl	mort_acc	tot_cur_bal	open_acc	pu
0	23.883190	2012	4.882877	59.321170	1.0	0.0	25.855653	9.0	
1	26.505820	2001	4.175401	117.019076	2.0	1.0	37.067491	11.0	
2	26.357777	1989	3.957849	92.352448	5.0	1.0	35.513545	15.0	
3	22.212451	2004	4.814081	77.290960	6.0	0.0	25.422894	12.0	
4	22.508416	1997	4.904883	54.443165	2.0	1.0	33.141233	8.0	

5 rows × 31 columns

11: Conclusion

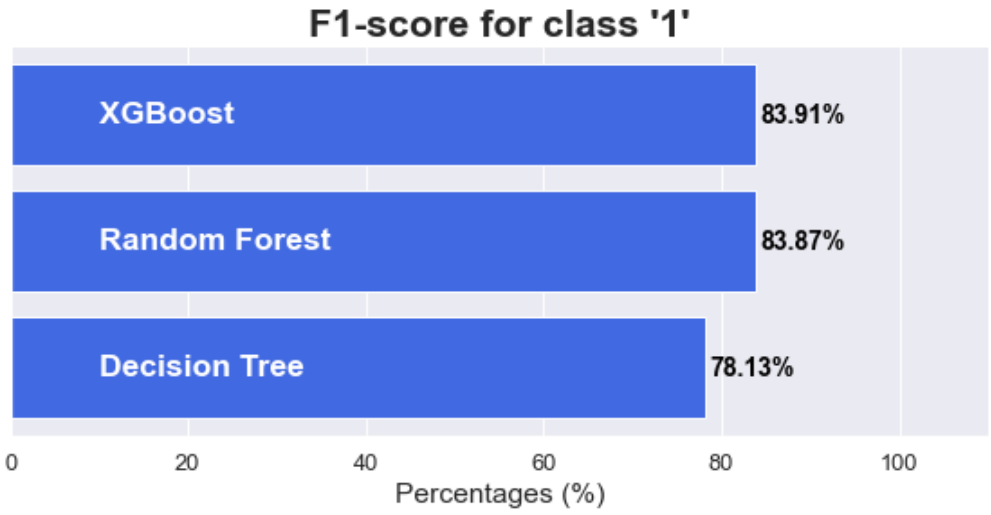
⬆️ Table Contents

In [127]:
```
results = pd.concat([dt_result, rf_result, xgb_result], axis=1).T

# Sort the dataframe in descending order based on F1-score (class 1) values
results.sort_values(by='F1-score (Class 1)', ascending=False, inplace=True)

# Color the F1-score column
results.style.applymap(lambda x: 'background-color: royalblue', subset='F1-score (Clas
```

Out[127]:

	Accuracy	Precision (Class 1)	Recall (Class 1)	F1-score (Class 1)	AUC (Class 1)
XGBoost	86.14%	98.3%	73.19%	83.91%	91.8%
Random Forest	85.28%	91.31%	77.55%	83.87%	91.82%
Decision Tree	78.68%	79.12%	77.17%	78.13%	79.9%

In [128]:
```
# Prepare values
results.sort_values(by='F1-score (Class 1)', ascending=True, inplace=True)
f1_scores = results['F1-score (Class 1)'].str.strip('%').astype(float)

# Plot the barh chart
fig, ax = plt.subplots(figsize=(10, 4), dpi=70)
ax.barh(results.index, f1_scores, color='royalblue')

# Annotate the values and indexes
for i, (value, name) in enumerate(zip(f1_scores, results.index)):
    ax.text(value+0.5, i, f"{value}%", ha='left', va='center', fontweight='bold', colo
    ax.text(10, i, name, ha='left', va='center', fontweight='bold', color='white', fon

# Remove yticks
ax.set_yticks([])

# Set x-axis limit
ax.set_xlim([0,110])

# Add title and xlabel
plt.title("F1-score for class '1'", fontweight='bold', fontsize=22)
plt.xlabel('Percentages (%)', fontsize=16)
plt.show()
```



Model Performance Evaluation - XGBoost

- 🏆 Accuracy: 86.14%
- 🏆 F1-score (Class 1): 83.91%
- 🏆 Precision (Class 1): 98.3%
- 🏆 Recall (Class 1): 73.91%
- 🏆 AUC (Class 1): 91.8%

Among all the tested classifiers, XGBoost demonstrated the best performance in predicting loan defaulters. With an accuracy of 86% and a high recall rate of 74%, the model effectively identifies instances of defaulters while maintaining a reasonable precision score.