ConsolidatedProject2

April 10, 2025

1 Machine Learning in Python - Project 2

Authors : Max Lawes, Mustafa Khan, Paras Ladwa

1.1 Setup

[1]:
```
# Data libraries
import pandas as pd
import numpy as np

# Plotting libraries
import matplotlib.pyplot as plt
import plotly.express as px
import seaborn as sns
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import PrecisionRecallDisplay
from sklearn.metrics import RocCurveDisplay


# sklearn modules
import sklearn
from sklearn.model_selection import train_test_split
from sklearn.impute import MissingIndicator
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.linear_model import BayesianRidge
from sklearn.pipeline import make_pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import FunctionTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, f1_score,␣
 ↪recall_score
from sklearn.metrics import roc_auc_score, classification_report
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder
from imblearn.pipeline import Pipeline
from imblearn.over_sampling import SMOTE
from imblearn.over_sampling import RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler
```

[2]:
```
!pip install imblearn
```

```
Requirement already satisfied: imblearn in /opt/conda/lib/python3.11/site-
packages (0.0)
Requirement already satisfied: imbalanced-learn in
/opt/conda/lib/python3.11/site-packages (from imblearn) (0.13.0)
Requirement already satisfied: numpy<3,>=1.24.3 in
/opt/conda/lib/python3.11/site-packages (from imbalanced-learn->imblearn)
(1.26.4)
Requirement already satisfied: scipy<2,>=1.10.1 in
/opt/conda/lib/python3.11/site-packages (from imbalanced-learn->imblearn)
(1.14.1)
Requirement already satisfied: scikit-learn<2,>=1.3.2 in
/opt/conda/lib/python3.11/site-packages (from imbalanced-learn->imblearn)
(1.5.2)
Requirement already satisfied: sklearn-compat<1,>=0.1 in
/opt/conda/lib/python3.11/site-packages (from imbalanced-learn->imblearn)
(0.1.3)
Requirement already satisfied: joblib<2,>=1.1.1 in
/opt/conda/lib/python3.11/site-packages (from imbalanced-learn->imblearn)
(1.4.2)
Requirement already satisfied: threadpoolctl<4,>=2.0.0 in
/opt/conda/lib/python3.11/site-packages (from imbalanced-learn->imblearn)
(3.5.0)
```

[3]:
```
# Plotting defaults,
plt.rcParams['figure.figsize'] = (8,6)
```

2 Introduction

For a mortgage company, the process of deciding if an applicant is suitable for a mortgage is a largely complex and important decision with potentially substantial ramifications. Being able to both make these decisions accurately is vital for a company to succeed and prosper.
Our report aims to utilise the provided data from Freddie Mac relating to both active and concluded mortgage agreements to inform and improve this process.

The provided dataset contains information regarding the clients such as credit score and first time homebuyer status, as well as features relating to the nature and procuration of the mortgages including the mortgage servicer, mortgage seller, interest rate and duration of the mortgage. This report presents a constructed model from the data, for the use of predicting which of the currently active mortgages provided by Freddie Mac are liable to defaulting. Through the implementation of the produced predictions of the model, we aim to minimise the risk harboured by Freddie Mac regarding clients defaulting.

Our process for constructing the final model included the exploration of the relationships between different features and their associated default rates, the construction of a baseline logistic regression, assessing the features considered relevant in predicting and culminating this information into producing a final random forest model.

The results of the model indicate that there are 866 active loans predicted to default before completion that may require your attention.

n.b. Throughout the report prepaid and defaulted are represented by 1 and 0 respectively.

3 Exploratory Data Analysis and Feature Engineering

We loaded our data in and dropped all entries relevant to active loans, as we are only concerened with loans which are either paid or defaulted. We also dropped certain columns from our analysis: `id_loan_rr`, `ppmt_pntly`, `prod_type`, and `io_ind`, as the ID columns have no bearing on our problem-solving, and the remaining columns have no variation between rows. Our data also contains multiple metrics for information such as location and loan time length. We drop `zipcodes` and `cd_msa` in favour of working solely with states, `st`, as our location indictor, while similarly dropping `dt_matr`, as we can calculate this information as needed from the `dt_first_pi` and `orig_loan_term`.

We have kept `id_loan` despite not incorporating it into our models, but due to the fact that we can use our final model to predict the status of active loans, `id_loan` will help the individuals to identify such ongoing cases.

[4]:
```python
# Load data in easyshare.csv
d = pd.read_csv("freddiemac.csv", low_memory=False)
# Create a working copy
df = d.copy()
```

[5]:
```python
# id column
df = df.drop(['id_loan_rr'] , axis = 1)

# no variation
df = df.drop(['ppmt_pnlty'] , axis = 1)
df = df.drop(['prod_type'] , axis = 1)
df = df.drop(['io_ind'] , axis = 1)
# we use st (state) instead
df = df.drop(['cd_msa'] , axis = 1)
df = df.drop(['zipcode'] , axis = 1)
# we already have beggining date and the length of loan, no need for this␣
 ↪column then
df = df.drop(['dt_matr'] , axis = 1)
```

A critical part of our preprocessing involves managing missing or unavailable information. In our dataset, unavailable data is often denoted by placeholder values—commonly some string of 9's.

While these placeholders serve as a flag for missingness, leaving them unaltered during model training can serverly corrupt and confuse our results. To make our data easier to work with for our model, we format these appropriately.

3.1 Handling NAs

[6]:
```
# as specified in documentation
df.loc[df['mi_pct'] == 999, 'mi_pct'] = np.nan
df.loc[df['cnt_units'] == 99, 'cnt_units'] = np.nan
df.loc[df['occpy_sts'] == str(9), 'occpy_sts'] = np.nan
df.loc[df['cltv'] == 999, 'cltv'] = np.nan
df.loc[df['ltv'] == 999, 'ltv'] = np.nan
df.loc[df['channel'] == str(9), 'channel'] = np.nan
df.loc[df['prop_type'] == str(99), 'prop_type'] = np.nan
df.loc[df['loan_purpose'] == str(9), 'loan_purpose'] = np.nan
df.loc[df['program_ind'] == str(9), 'program_ind'] = np.nan
df.loc[df['property_val'] == 9, 'property_val'] = np.nan
```

With the standardisation, we must also adjust the variables 'flag_sc' and 'rr_ind' with binary data which utilise NaN to encode a 'No' response as to not misinterpret values as missing data.

[7]:
```
df['flag_sc'] = df['flag_sc'].fillna(value = 'N')
df['rr_ind'] = df['rr_ind'].fillna(value = 'N')
```

For the variable `fico`, which contains the individuals credit score, the value "9999" represents values which fall outside of the range [300,850]. As such, imputing the data is not applicable as the new values would be extrapolated out of the range of data points and negatively affect the model, so these points are removed.
This process is also needed to be applied to `dti`, for which "999" encodes values greater than 65.

[8]:
```
df = df[df['fico'] != 9999]
df = df[df['dti'] != 999]
```

We have also identified that the `property_val` variable is a categorical variable encoded numerically. For this, we relabelled the inputs to what they represent appropriately.

[9]:
```
df['property_val'] = df['property_val'].replace({1: 'ace_loans', 2:␣
 ↪'full_appraisal', 3: 'other_appraisal', 4: 'ace_pdr'})
df['property_val'] = df['property_val'].fillna('full_appraisal')
```

We next investigate any missing values or entries from our data. The table below indicates how many entries are missing from each variable. In particular, we are missing an entry from each of `mi_pct`, 2 from `ltv`, 6 from `cltv`, and 181732 from `program_ind`. We use an iterative imputer to transform the columns and fill in the missing entries, for all variables but `program_ind`. Due to the severe scarcity of entries for this variable, we cannot justify making significant inferences from it, therefore we ignore it for the remainder of our analysis and choose not to include it in our baseline model.

[10]:
```
# Calculate number and percentage of missing values
missing_counts = df.isna().sum()
missing_percent = (df.isna().mean() * 100).round(2)

# Combine into a DataFrame and filter only columns with missing data
missing_table = pd.DataFrame({
    'Missing Count': missing_counts,
    'Missing %': missing_percent
})

# Filter to show only columns with at least one missing value
missing_table = missing_table[missing_table['Missing Count'] > 0]

# Display the table
print(missing_table)
```

```
             Missing Count  Missing %
mi_pct                   1       0.00
cltv                     6       0.00
ltv                      2       0.00
program_ind         181732      91.99
```

[11]:
```
df = df.drop(['program_ind'] , axis = 1)
```

[12]:
```
#identified variables to impute
cols_to_impute = ['mi_pct', "cltv", "ltv"]
# Create the imputer
imputer = IterativeImputer(estimator=BayesianRidge(), max_iter=20,␣
 ↪random_state=42)

# Fit and transform only the selected columns
imputed_array = imputer.fit_transform(df[cols_to_impute])

# Replace the imputed columns in the original DataFrame
df = df.copy()
df[cols_to_impute] = imputed_array
```

```
/opt/conda/lib/python3.11/site-packages/sklearn/impute/_iterative.py:825:
ConvergenceWarning: [IterativeImputer] Early stopping criterion not reached.
  warnings.warn(
```

We choose to include two temporal features; `orig_loan_term` and `dt_first_pi`. The former is already in the format required. The latter is of the form $YYYYMM$ which we simply have converted to number of months by $YYYY \cdot 12 + MM$, after this each date is subtracted by the smallest of the dates. We redefine this variable as `initial_month` which indicates number of months since the first loans initial date.

[13]:
```
def fix_dates(dates):

    new = np.empty(len(dates))
    dates = np.array(dates)

    for i in range(len(dates)):

        d = dates[i]

        yyyy = str(d)[:4]
        mm = str(d)[4:]
        yyyy, mm = float(yyyy), float(mm)

        v = yyyy*12 + mm
        new[i] = v

    return new - min(new)

df['initial_month'] = np.array(fix_dates(df['dt_first_pi']), dtype = int)
df = df.drop(['dt_first_pi'], axis = 1)
```

With our preprocessing complete we separate the active loans out of our dataset as our model is only concerned with defaulted or prepaid loans. Upon completion of our final model we use the active loans data to predict whether the cases will be flagged by our model or not.

We divide the data, keeping 30% of it to test our model on, after training it on the remainder of the data. This helps us test any potential model bias, or overfitting from our data. We set a seed before doing so in order to ensure reproducibility of results. We separate `loan_status`, the target variable we wish to predict from the remainder of the data, and summarize the proportion of default loans in both the test and training set. We see that for both of them the proportion is roughly 0.59%, indicating that loan defaults in general make up a small fraction of our data. This heavily influences our modelling decisions and is depicted further in the report.

We would like to note that we drop our target variable, loan status from the test and train data, *immediately after* we split it, in order to create a copy of X_train for EDA purposes. The test and training sets used for the model do not contain the target variable, so we can confirm the data has not been compromised.

[14]:
```
df_active = df[df['loan_status'] == 'active']

count = df[df['loan_status'] == 'active'].shape[0]
df = df[df['loan_status'] != 'active']

X = df.copy()

y = df['loan_status']
y = LabelEncoder().fit_transform(y) #prepaid = 1, default = 0
rng = np.random.seed(0)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,␣
 ↪stratify=y,
                                                    random_state = rng)

X_vis = X_train.copy()
X_train = X_train.drop(['loan_status'], axis = 1)
X_test = X_test.drop(['loan_status'], axis = 1)

print("Proportion within training data")
print(pd.Series(y_train).value_counts(normalize=True)*100)
print("Proportion within testing data")
print(pd.Series(y_test).value_counts(normalize=True)*100)
```

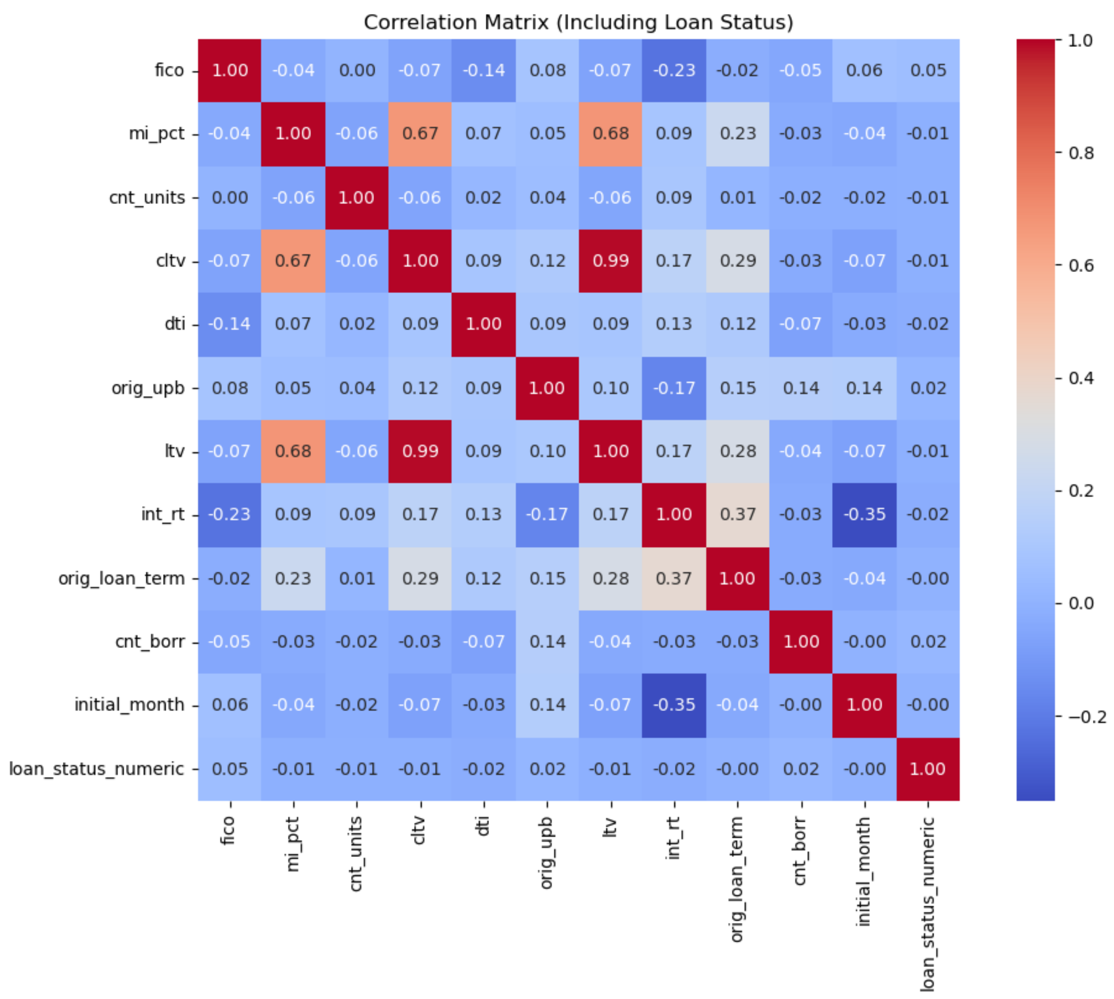
```
Proportion within training data
1    99.43712
0     0.56288
Name: proportion, dtype: float64
Proportion within testing data
1    99.43638
0     0.56362
Name: proportion, dtype: float64
```

Due to significantly unbalanced data we note that it is vital that we use stratify when we splitting into training and testing sets. This enables that the proportions between the datasets are equal, to ensure that the results produced are reliable and representative.

3.2 Describing Data

We generated a heatmap to investigate the correlations between variables, and plot their distributions as well. We see the highest correlation between `mi_pct` and both the loan to value variables (`cltv` and `ltv`).

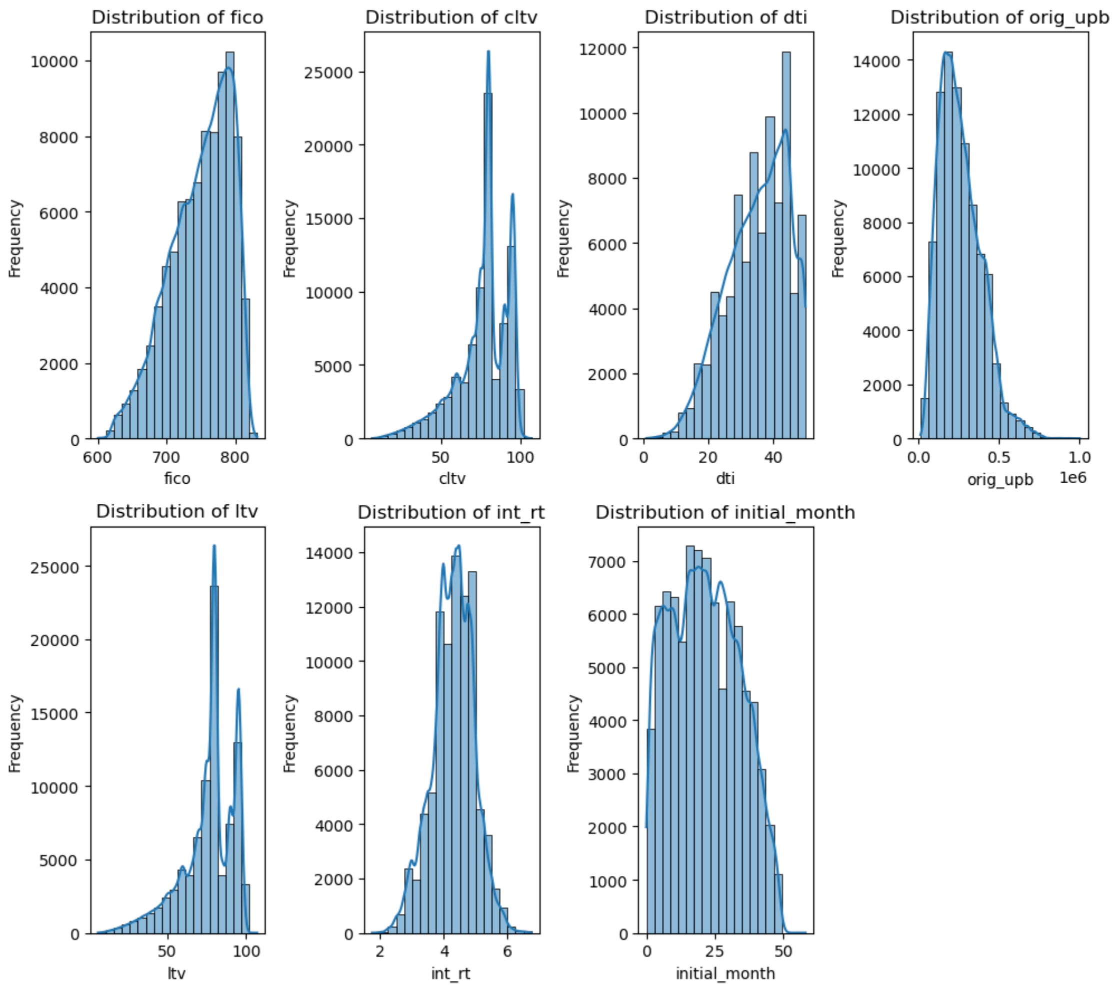
[15]:
```
heatmap_data = X_train.copy()
heatmap_data['loan_status_numeric'] = df['loan_status'].map({'default': 0,␣
 ↪'prepaid': 1})

# Compute correlation matrix
corr = heatmap_data.select_dtypes(include='number').corr()

# Plot
plt.figure(figsize=(12, 8))
sns.heatmap(corr, annot=True, fmt=".2f", cmap='coolwarm', square=True)
plt.title("Correlation Matrix (Including Loan Status)")
plt.show()
```



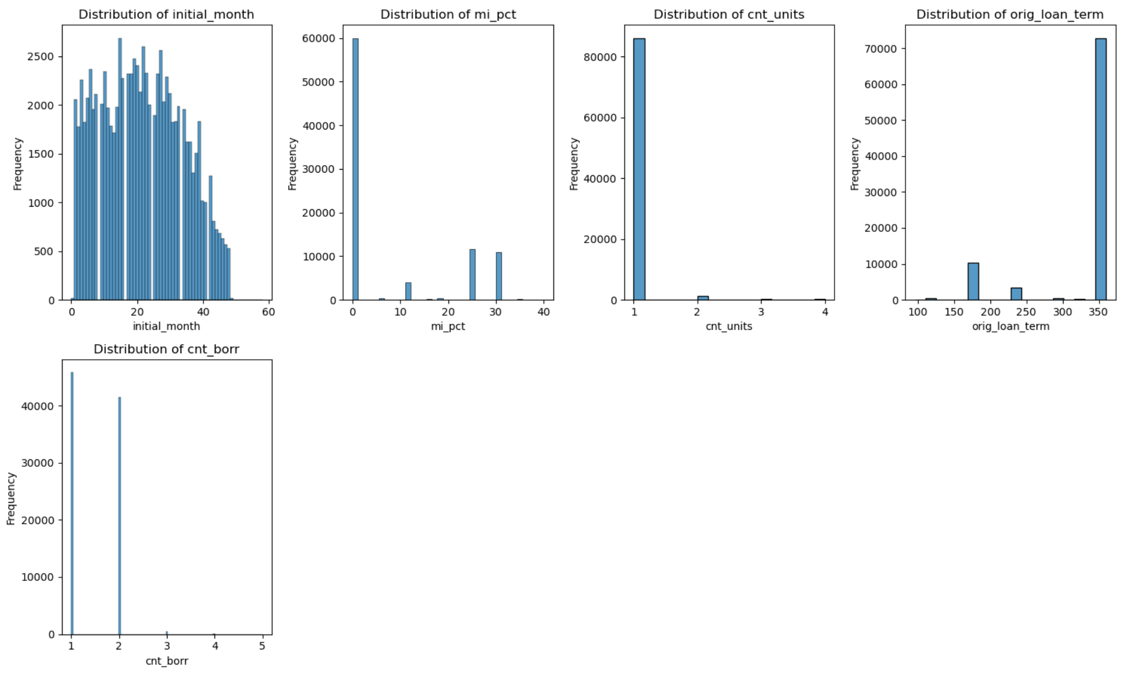
	fico	mi_pct	cnt_units	cltv	dti	orig_upb	ltv	int_rt	orig_loan_term	cnt_borr	initial_month	loan_status_numeric
fico	1.00	-0.04	0.00	-0.07	-0.14	0.08	-0.07	-0.23	-0.02	-0.05	0.06	0.05
mi_pct	-0.04	1.00	-0.06	0.67	0.07	0.05	0.68	0.09	0.23	-0.03	-0.04	-0.01
cnt_units	0.00	-0.06	1.00	-0.06	0.02	0.04	-0.06	0.09	0.01	-0.02	-0.02	-0.01
cltv	-0.07	0.67	-0.06	1.00	0.09	0.12	0.99	0.17	0.29	-0.03	-0.07	-0.01
dti	-0.14	0.07	0.02	0.09	1.00	0.09	0.09	0.13	0.12	-0.07	-0.03	-0.02
orig_upb	0.08	0.05	0.04	0.12	0.09	1.00	0.10	-0.17	0.15	0.14	0.14	0.02
ltv	-0.07	0.68	-0.06	0.99	0.09	0.10	1.00	0.17	0.28	-0.04	-0.07	-0.01
int_rt	-0.23	0.09	0.09	0.17	0.13	-0.17	0.17	1.00	0.37	-0.03	-0.35	-0.02
orig_loan_term	-0.02	0.23	0.01	0.29	0.12	0.15	0.28	0.37	1.00	-0.03	-0.04	-0.00
cnt_borr	-0.05	-0.03	-0.02	-0.03	-0.07	0.14	-0.04	-0.03	-0.03	1.00	-0.00	0.02
initial_month	0.06	-0.04	-0.02	-0.07	-0.03	0.14	-0.07	-0.35	-0.04	-0.00	1.00	-0.00
loan_status_numeric	0.05	-0.01	-0.01	-0.01	-0.02	0.02	-0.01	-0.02	-0.00	0.02	-0.00	1.00

The remaining data is then plotted on histograms to examine the key features of both our categorical and numerical variables. For each variable, we pay close attention to the distribution and any inherent properties that may affect model performance. In particular, our numerical variables `fico`, `cltv`, and `dti` seem to be negatively skewed, with repect to their distribution, while `orig_upb` is more positively skewed. In our catagorical variables, we noticed that the distribution of `cnt_units` is far more unbalanced than the other variables, with an overwhelming majority of it being 1. Recognising the imbalance this variable would present in our model, we chose to disregard it from our baseline.

[16]:
```
# List of numerical features you want to plot
features_num = ['fico', 'cltv', 'dti', 'orig_upb', 'ltv', 'int_rt', ␣
 ↪'initial_month']

x_labels = [f"Distribution of {feature}" for feature in features_num]

plt.figure(figsize=(10, 9))
for i, feature in enumerate(features_num):
    plt.subplot(2, 4, i+1)
    sns.histplot(X_train[feature], kde=True, bins = 20)
    plt.title(x_labels[i])
    plt.ylabel('Frequency')

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



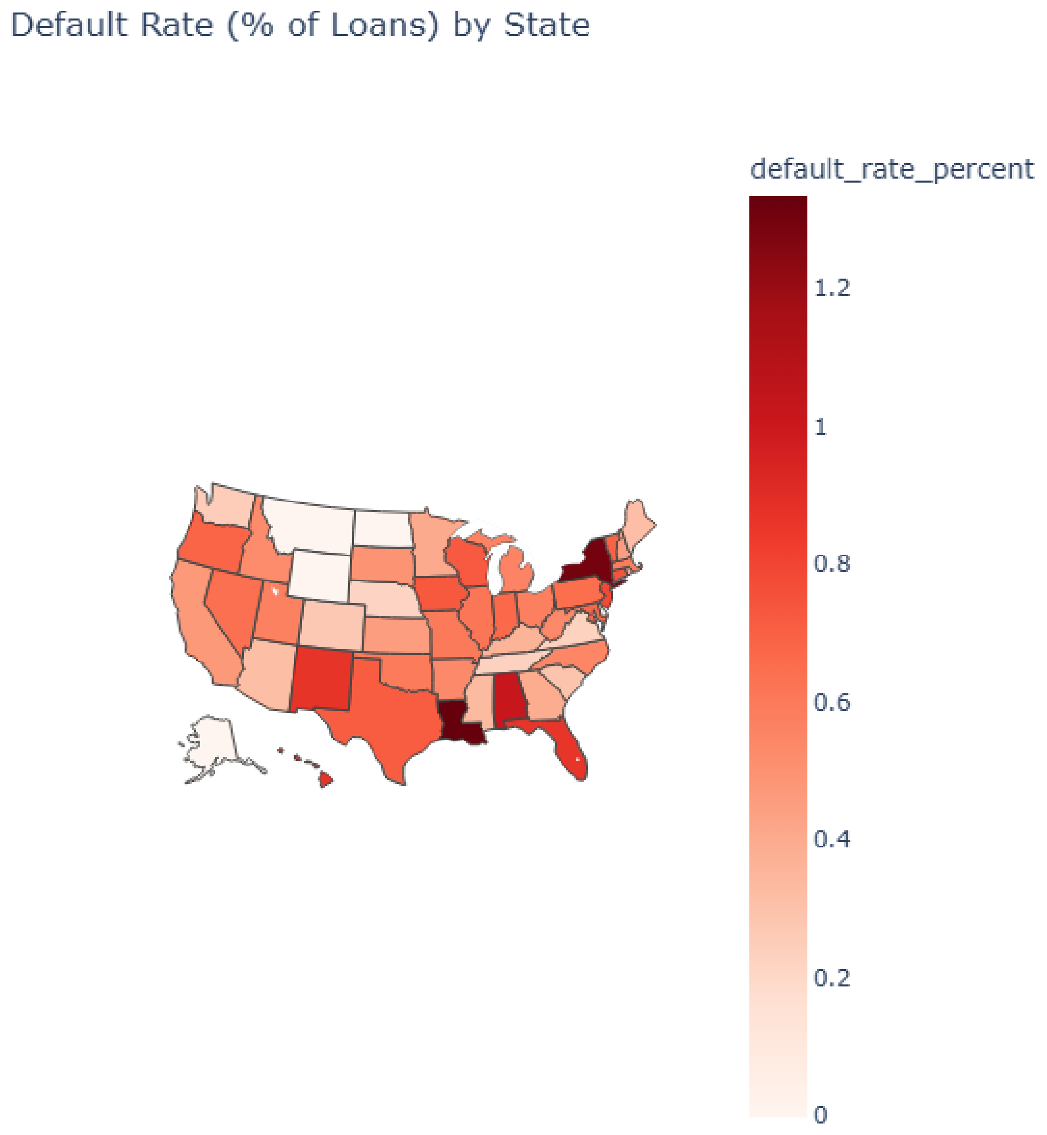
```
[17]: # List of categorical or numerical features to plot
features_cat = ['initial_month', 'mi_pct', 'cnt_units', 'orig_loan_term',␣
 ↪'cnt_borr']
x_labels = [f"Distribution of {feature}" for feature in features_cat]

plt.figure(figsize=(15, 9))
for i, feature in enumerate(features_cat):
    plt.subplot(2, 4, i+1)
    sns.histplot(X_train[feature])
    plt.title(x_labels[i])
    plt.ylabel('Frequency')

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



With geographic location often having an affect on one's quality of life, and economic status, we generated a heatmap to examine average default rates of each state of the US. Our map's output supports this, with there being signigicant variation in default rates between states, the highest of which was 1.3% in Louisiana.

```
[18]: total_loans = X_vis['st'].value_counts().reset_index()
total_loans.columns = ['state', 'total_loans']

defaults = X_vis[X_vis['loan_status'] == 'default']
state_defaults = defaults['st'].value_counts().reset_index()
state_defaults.columns = ['state', 'defaults']

merged = pd.merge(total_loans, state_defaults, on='state', how='left')
merged['defaults'] = merged['defaults'].fillna(0)  # in case some states have␣
 ↪no defaults
merged['default_rate_percent'] = (merged['defaults'] /␣
 ↪merged['total_loans'])*100
merged['default_rate_percent'] = merged['default_rate_percent'].round(3)
```

```python
fig = px.choropleth(
    merged,
    locations='state',
    locationmode='USA-states',
    color='default_rate_percent',
    scope='usa',
    color_continuous_scale='reds',
    title='Default Rate (% of Loans) by State',
    width=1000,
    height=700
)

fig.show()
#format-  explain this more
```

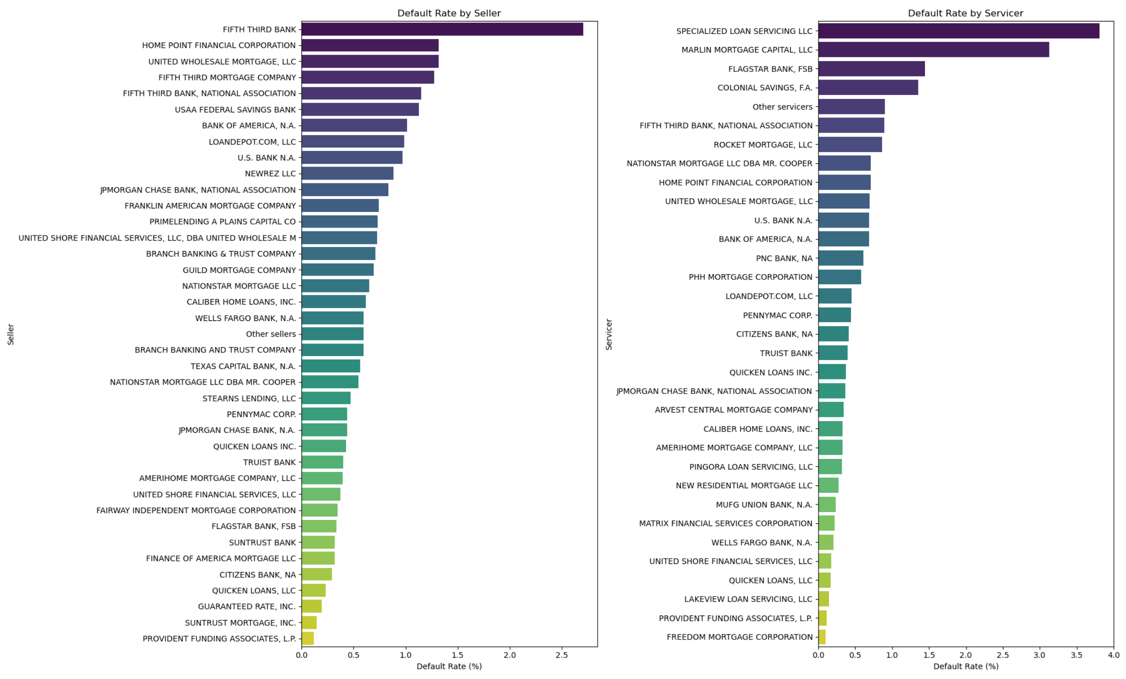


The two plots below denote the default rates, with respect to mortgage sellers and servicing companies. We observe similar variation as in the heatmap, with different servicers and sellers having varying default rate revels. We could attribute this too poor management or buisness practicies within these companies. For instance, one of the highest default rates in the servicer plot corresponds to `other_servicers`, meaning we have no information to attest to the reliability of these companies, or their size or capabilities

[19]:
```
total_loans = X_vis['seller_name'].value_counts().reset_index()
total_loans.columns = ['seller', 'total_loans']

defaults = X_vis[X_vis['loan_status'] == 'default']
seller_defaults = defaults['seller_name'].value_counts().reset_index()
seller_defaults.columns = ['seller', 'defaults']

seller_df = pd.merge(total_loans, seller_defaults, on='seller', how='left')
seller_df['defaults'] = seller_df['defaults'].fillna(0)  # in case some sellers␣
 ↪have no defaults
seller_df['default_rate_percent'] = (seller_df['defaults'] /␣
 ↪seller_df['total_loans'])*100
seller_df['default_rate_percent'] = seller_df['default_rate_percent'].round(3)
seller_df = seller_df.sort_values(by='default_rate_percent', ascending=False)

total_loans3 = X_vis['servicer_name'].value_counts().reset_index()
total_loans3.columns = ['servicer', 'total_loans']

defaults3 = X_vis[X_vis['loan_status'] == 'default']
servicer_defaults = defaults3['servicer_name'].value_counts().reset_index()
servicer_defaults.columns = ['servicer', 'defaults']

servicer_df = pd.merge(total_loans3, servicer_defaults, on='servicer',␣
 ↪how='left')
servicer_df['defaults'] = servicer_df['defaults'].fillna(0)  # in case some␣
 ↪sellers have no defaults
servicer_df['default_rate_percent'] = (servicer_df['defaults'] /␣
 ↪servicer_df['total_loans'])*100
servicer_df['default_rate_percent'] = servicer_df['default_rate_percent'].
 ↪round(3)
servicer_df = servicer_df.sort_values(by='default_rate_percent',␣
 ↪ascending=False)

# Filter out entries with 0 default rate for sellers and servicers
seller_df_filtered = seller_df[seller_df['default_rate_percent'] > 0]
servicer_df_filtered = servicer_df[servicer_df['default_rate_percent'] > 0]

# Create one figure with two subplots side-by-side
fig, axes = plt.subplots(1, 2, figsize=(20, 12))

# Plot for sellers
sns.barplot(data=seller_df_filtered, x='default_rate_percent', y='seller',␣
 ↪palette='viridis', ax=axes[0])
axes[0].set_xlabel('Default Rate (%)')
axes[0].set_ylabel('Seller')
axes[0].set_title('Default Rate by Seller')
```

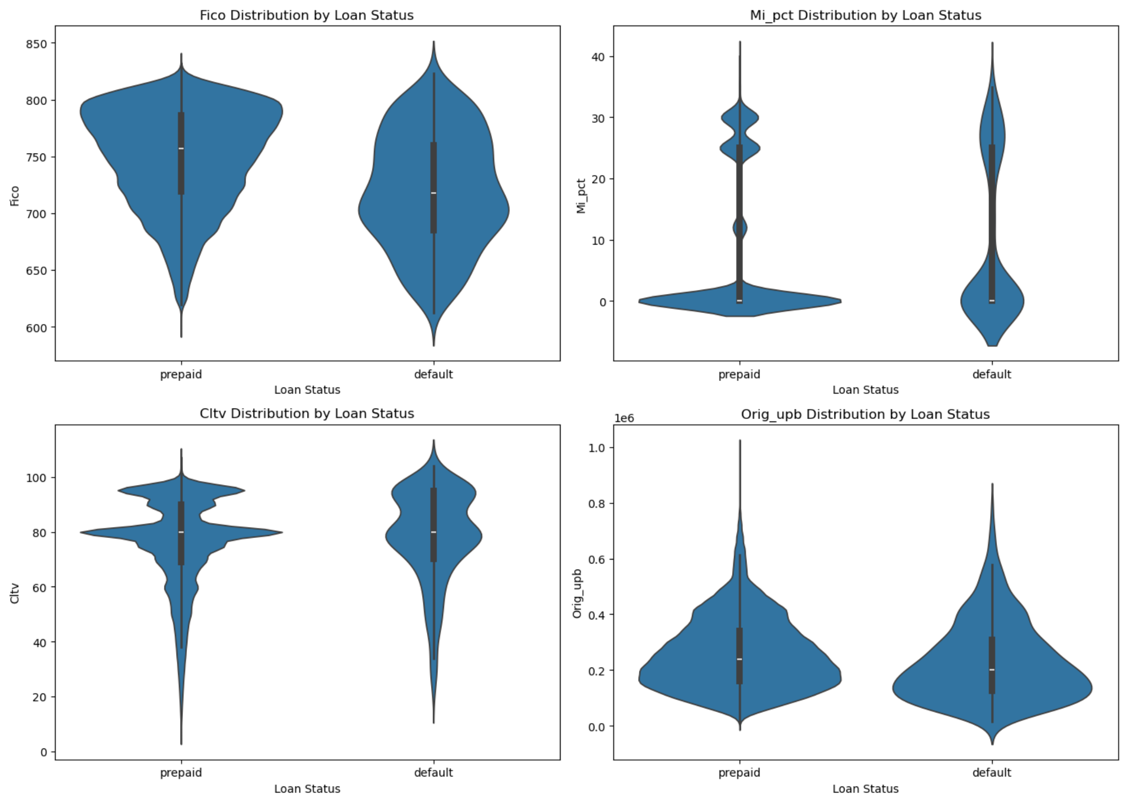
```
# Plot for servicers
sns.barplot(data=servicer_df_filtered, x='default_rate_percent', y='servicer',␣
 ↪palette='viridis', ax=axes[1])
axes[1].set_xlabel('Default Rate (%)')
axes[1].set_ylabel('Servicer')
axes[1].set_title('Default Rate by Servicer')

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

```
/tmp/ipykernel_26528/173392342.py:35: FutureWarning:



Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `y` variable to `hue` and set `legend=False` for the same
effect.


/tmp/ipykernel_26528/173392342.py:41: FutureWarning:



Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `y` variable to `hue` and set `legend=False` for the same
effect.
```



Here we visualise distributions of certain features of the training data between prepaid and default cases. We can see that these variables typically share similar distributions between prepaid and default, except for `fico`.

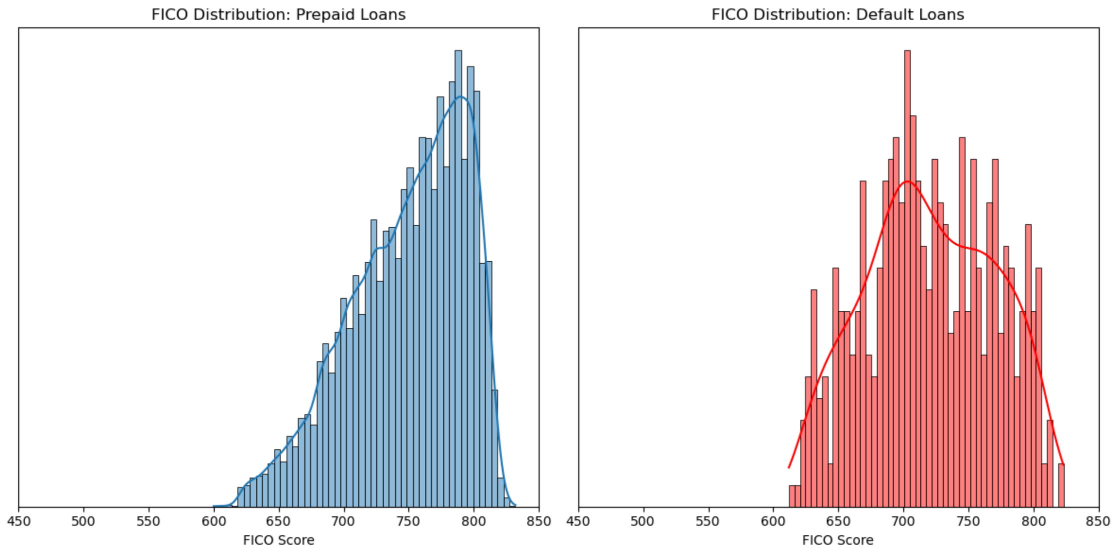
[20]:
```python
y_vars = ['fico', 'mi_pct', 'cltv', 'orig_upb']

# Create a 2x2 grid of subplots; adjust figsize as needed
fig, axes = plt.subplots(2, 2, figsize=(14, 10))
axes = axes.flatten()  # flatten the grid so you can iterate easily

# Loop through each variable and plot a violin plot
for ax, var in zip(axes, y_vars):
    sns.violinplot(x='loan_status', y=var, data=X_vis, inner='box', ax=ax)
    ax.set_title(f'{var.capitalize()} Distribution by Loan Status')
    ax.set_xlabel('Loan Status')
    ax.set_ylabel(var.capitalize())

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



Our interpretation of the `fico`'s distribution, as well as violin plot suggested that it's distribution demanded futher examination. We therefore plotted the distribution of the `fico` scores for defaulted loans and prepaid loans seperately, to examine the variable through a deeper lens. We can see almost opposing skews below for the two plots, with the defaulted loans tending towards lower `fico` scores, and the prepaid loans tending to higher ones. This is in line with what we would expect, as a low credict score shoud indicate low fincancial trustworthiness.

[21]:
```
# Filter the dataframe by loan status
df_prepaid = X_vis[X_vis['loan_status'] == 'prepaid']
df_default = X_vis[X_vis['loan_status'] == 'default']

# Set up the subplots: one row, two columns
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

# Histogram for prepaid loans
sns.histplot(data=df_prepaid, x='fico', bins=50, ax=axes[0], kde=True)
axes[0].set_title("FICO Distribution: Prepaid Loans")
axes[0].set_xlabel("FICO Score")
axes[0].set_ylabel("Number of Loans")
axes[0].set_xlim(450, 850)
axes[0].set_ylabel('')
axes[0].set_yticks([])
```

```
# Histogram for default loans
sns.histplot(data=df_default, x='fico', bins=50, ax=axes[1],color = 'red',␣
    ↪kde=True)
axes[1].set_title("FICO Distribution: Default Loans")
axes[1].set_xlabel("FICO Score")
axes[1].set_xlim(450, 850)
axes[1].set_ylabel('')
axes[1].set_yticks([])

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



4 Model Fitting and Tuning

Here we explore the models used in this report, We began by establishing a baseline model which was a simple Logistic regression model. We included all features which provided further information to whether the loan was to default or not. This was a simple and interpretable place for us to begin and proved successful in predictions, specifically with proportions as this was one of our primary objectives.

Beyond this we evaluated a few other models, primarily exploring Random Forests with and without, random undersampling and random oversampling. Below we discuss the results from this.

A metric we considered throughout the modelling is the FP FN tradeoff. A FP represents the model predicting prepaid where the true outcome is default. A FN is a predicted default, though the actual outcome is the mortgage is prepaid. A FN is much prefered as this represents a client being refused a mortgage they would have prepaid, resulting in a loss of potential profit. FN looses income though FP looses capital as the bank would have to cover the costs of the defaulted case which is a financial loss for Freddie Mac.

5 Baseline - Logistic Regression with RandomOverSampler

Here we initialise and fit our baseline, a key componenent to this is the `RandomOverSampler`, this oversamples the minority class (defaulted cases) by randomly selecting samples with replacement. This process allows us to take into account the imbalanced nature of the source data and work around it. Here we also include as many features as possible such that we can visualise the prevalant ones.

[22]:
```
# Preprocessor (as before)
def map_Y_to_1(X):
    return (X == 'Y').astype(int)

preprocessor = ColumnTransformer([
    ('flag_sc', FunctionTransformer(map_Y_to_1,␣
 ↪feature_names_out='one-to-one'), ['flag_sc']),
    ('rr_ind', FunctionTransformer(map_Y_to_1, feature_names_out='one-to-one'),␣
 ↪['rr_ind']),
    ('flag_fthb', OneHotEncoder(drop=["N"]), ['flag_fthb']),
    ('categoricals', OneHotEncoder(), [
        'occpy_sts', 'property_val', "channel", "st", "prop_type",
        "loan_purpose", "seller_name", "servicer_name", "mi_cancel_ind"
    ]),
    ('numeric', StandardScaler(), [
        'fico', 'mi_pct', 'cltv', 'dti', 'orig_upb', 'ltv',
        'int_rt', 'orig_loan_term', 'cnt_borr', 'initial_month'
    ])
])

# Full pipeline: preprocess → oversample → model
log_pipe = Pipeline([
    ("preprocessing", preprocessor),
    ("oversample", RandomOverSampler(random_state=42)),
    ("classifier", LogisticRegression(random_state=42, penalty=None,␣
 ↪max_iter=1000, class_weight='balanced'))
])

log_pipe.fit(X_train, y_train)
```

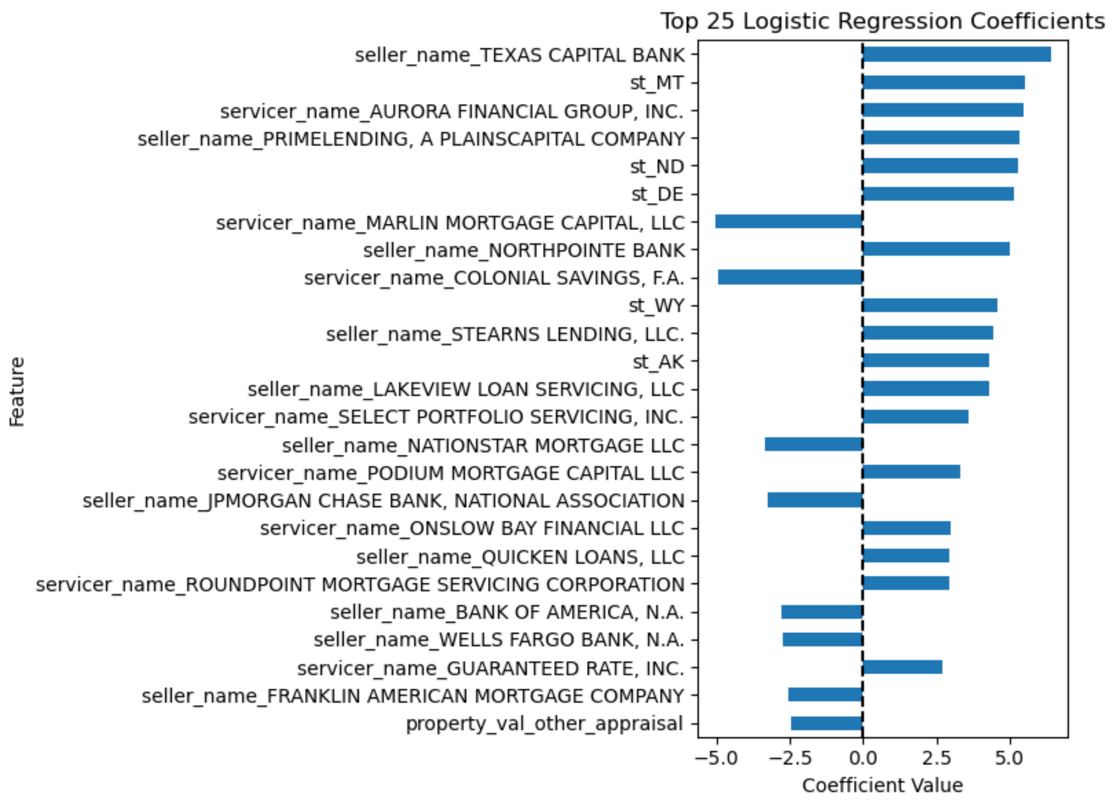
[22]:
```
Pipeline(steps=[('preprocessing',
                 ColumnTransformer(transformers=[('flag_sc',
FunctionTransformer(feature_names_out='one-to-one',
func=<function map_Y_to_1 at 0x7ffa4a58c7c0>),
                                                  ['flag_sc']),
                                                 ('rr_ind',
FunctionTransformer(feature_names_out='one-to-one',
func=<function map_Y_to_1 at 0x7ffa4a58c7c0>),
                                                  ['rr_ind']),
                                                 ('flag_fthb',
                                                  OneHotEncoder(drop=['N']),
                                                  ['flag_fthb'…
                                                   'channel', 'st', 'prop_type',
                                                   'loan_purpose',
                                                   'seller_name',
                                                   'servicer_name',
                                                   'mi_cancel_ind']),
                                                 ('numeric', StandardScaler(),
                                                  ['fico', 'mi_pct', 'cltv',
                                                   'dti', 'orig_upb', 'ltv',
                                                   'int_rt', 'orig_loan_term',
                                                   'cnt_borr',
                                                   'initial_month'])])),
                ('oversample', RandomOverSampler(random_state=42)),
                ('classifier',
                 LogisticRegression(class_weight='balanced', max_iter=1000,
                                    penalty=None, random_state=42))])
```

Here we develop a plot of the top 25 Logistic regression coefficients, this provides a ranking of features which are most prevelant the baseline model uses to predict defaulted/prepaid data.

[23]:
```python
# Updated step names
preprocessor = log_pipe.named_steps['preprocessing']
model = log_pipe.named_steps['classifier']

def get_feature_names(column_transformer):
    feature_names = []

    for name, trans, cols in column_transformer.transformers_:
        if name == 'remainder' and trans == 'drop':
            continue
        if hasattr(trans, 'get_feature_names_out'):
            names = trans.get_feature_names_out(cols)
        elif hasattr(trans, 'get_feature_names'):
            names = trans.get_feature_names(cols)
        else:
            names = cols  # passthrough
        feature_names.extend(names)

    return feature_names

feature_names = get_feature_names(preprocessor)

# Create coefficient DataFrame
```

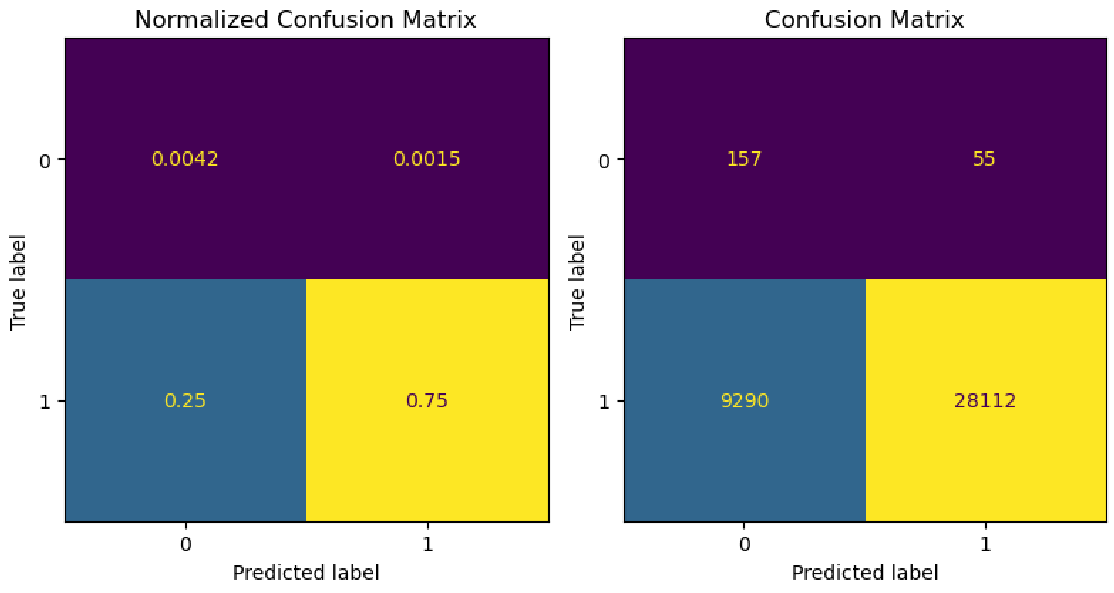
```
coefs = pd.DataFrame({
    "Feature": feature_names,
    "Coefficient": model.coef_.flatten()
})

coefs_sorted = coefs.reindex(coefs.Coefficient.abs().
 ↪sort_values(ascending=False).index)

plt.figure(figsize=(12, 10))  # Adjust size as needed
coefs_sorted.head(25).plot.barh(x='Feature', y='Coefficient', legend=False)
plt.title("Top 25 Logistic Regression Coefficients")
plt.axvline(0, color='black', linestyle='--')
plt.xlabel("Coefficient Value")
plt.gca().invert_yaxis()
plt.tight_layout()
plt.show()
```

```
<Figure size 1200x1000 with 0 Axes>
```



The results of our coefficient plot is significantly dominated by a few features: `seller_name`, `servicer_name` and `st` (representing state) to a lesser extent. Although we expected `fico` (representing credit score) to be the major contribution here, this gives as a good idea of other features which we should implement into our final model.

```
[24]: # Create a 2x1 subplot
fig, axes = plt.subplots(1, 2, figsize=(8, 10))

# First Confusion Matrix (normalized)
ConfusionMatrixDisplay.from_estimator(log_pipe, X_test, y_test, colorbar=False,␣
 ↪normalize='all', ax=axes[0])
axes[0].set_title('Normalized Confusion Matrix')

# Second Confusion Matrix (not normalized)
ConfusionMatrixDisplay.from_estimator(log_pipe, X_test, y_test, colorbar=False,␣
 ↪ax=axes[1])
axes[1].set_title('Confusion Matrix')

# Display the plot
plt.tight_layout()
plt.show()
```



Above we have the confusion matrix of our baseline models performance, where 1 represents *prepaid* and 0 represents *defaulted*. If we recall the proportion of these from our training dataset we had ~0.99 and ~0.50 but here we have ~0.75 and ~0.25 defaulted and prepaid respectively. It is evident our model is flawed in this perspective though we opted for this over others due to the comprable nature with our final model.

```
[25]: y_pred = log_pipe.predict(X_test)

print("Logistic Regression Evaluation Metrics")
print("-" * 40)
print(f"Accuracy         : {accuracy_score(y_test, y_pred):.4f}")
print(f"Precision (Default) : {precision_score(y_test, y_pred, pos_label=1):.
 ↪4f}")
print(f"Recall (Default)    : {recall_score(y_test, y_pred, pos_label=1):.4f}")
print(f"F1 Score (Default)  : {f1_score(y_test, y_pred, pos_label=1):.4f}")
print("-" * 40)
```

```
Logistic Regression Evaluation Metrics
----------------------------------------
Accuracy         : 0.7516
Precision (Default) : 0.9980
Recall (Default)    : 0.7516
F1 Score (Default)  : 0.8575
----------------------------------------
```

Precision is defined as :

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

where TP and FP are True Positives and False Positives. TP represents a loan predicted as prepaid which is in fact prepaid, where FP represents a loan predicted prepaid though the loan is defaulted.

The precision is the proportion of predicted prepaids which are correctly predicted. It then follows that 1 - Precision = 0.002 represents the proportion of predicted prepaids which default. Refering to the tradeoff described above, this is preffered as there are fewer cases where the model will predict this such that the bank ends up paying for the defaulted cases.

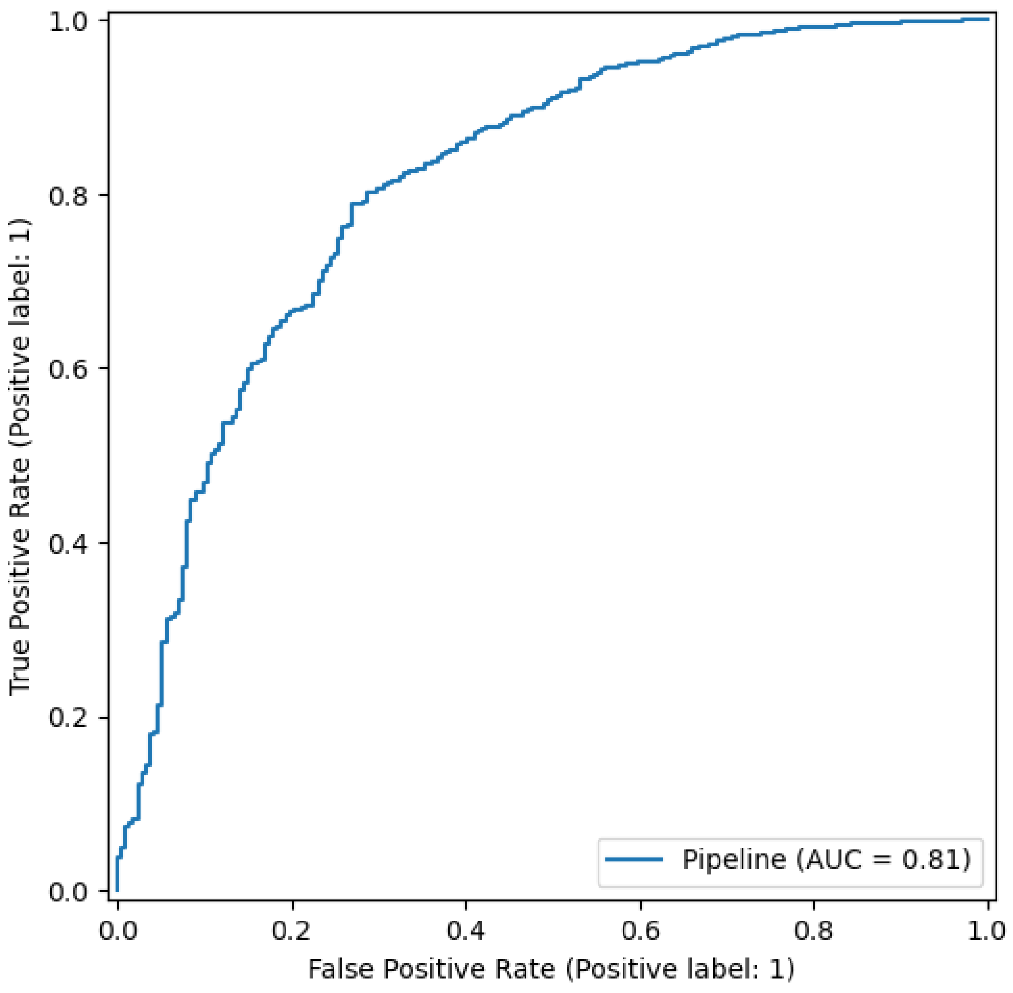
Recall is defined as :

$$\text{Recall (TPR)} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Where FN is False Negatives (predicted default, though prepaid).

Here our Recall rate indicates our baseline model is more cautious - giving us a high FN rate.

[26]:
```
RocCurveDisplay.from_estimator(log_pipe, X_test, y_test)
plt.show()

# Compute the AUC
y_test_probs = log_pipe.predict_proba(X_test)[:, 1]
```



Above we have can see our ROC curve hugs the top left corner, this indicates a high True Positive rate (TPR) and a low False Positive rate (FPR) as we wanted for our model, in particular we want a low FPR as a high FPR would cost the bank most and lead to granting mortgages to individuals who are likely to default.

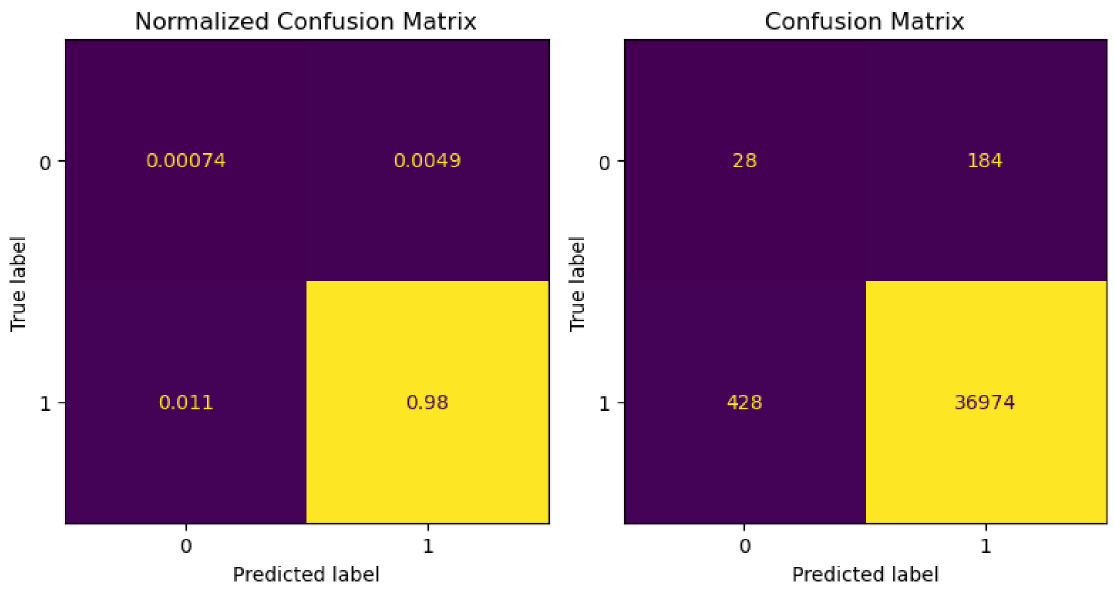
6 Final Model with Oversampling

Below we develop our final model, again implementing `RandomOverSampler` addressing the imbalance. We include fewer features here to keep complexity low to reduce computational costs. As per our baseline we include the three most prominent features `state`, `seller_name`, and `servicer_name`. Beyond these we have also chosen to include `fico` due to its inherent nature in any financed purchase along side our EDA findings.

[27]:
```
rf_clf_over = Pipeline([
    ('preprocessor', ColumnTransformer([
        ('categoricals', OneHotEncoder(), ["seller_name", "servicer_name",␣
↪'st']),
        ('numericals', StandardScaler(), ['fico'])
    ])),
    ('oversampler', RandomOverSampler(random_state=42)),  # oversampling step
    ('classifier', RandomForestClassifier(n_estimators=10, random_state=42,␣
↪class_weight='balanced'))
])

rf_clf_over.fit(X_train, y_train)
```

[27]:
```
Pipeline(steps=[('preprocessor',
                 ColumnTransformer(transformers=[('categoricals',
                                                  OneHotEncoder(),
                                                  ['seller_name',
                                                   'servicer_name', 'st']),
                                                 ('numericals',
                                                  StandardScaler(),
                                                  ['fico'])])),
                ('oversampler', RandomOverSampler(random_state=42)),
                ('classifier',
                 RandomForestClassifier(class_weight='balanced',
                                        n_estimators=10, random_state=42))])
```

[28]:
```
# Create a 2x1 subplot
fig, axes = plt.subplots(1, 2, figsize=(8, 10))

# First Confusion Matrix (normalized)
ConfusionMatrixDisplay.from_estimator(rf_clf_over, X_test, y_test,␣
↪colorbar=False, normalize='all', ax=axes[0])
axes[0].set_title('Normalized Confusion Matrix')

# Second Confusion Matrix (not normalized)
ConfusionMatrixDisplay.from_estimator(rf_clf_over, X_test, y_test,␣
↪colorbar=False, ax=axes[1])
axes[1].set_title('Confusion Matrix')

# Display the plot
plt.tight_layout()
plt.show()
```



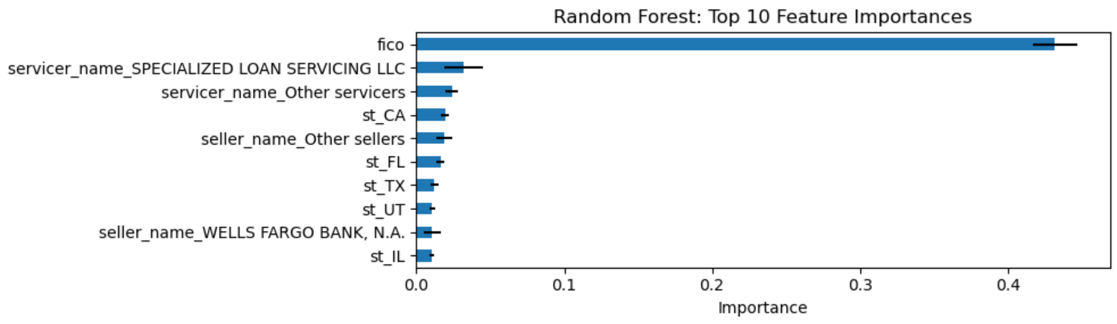
In contrast to the confusion matrix of our baseline, we can see the balance far more mirrors that of the original data.

```
[29]: y_pred = rf_clf_over.predict(X_test)
      accuracy = accuracy_score(y_test, y_pred)
      precision = precision_score(y_test, y_pred, average='weighted')
      recall = recall_score(y_test, y_pred, average='weighted')
      f1 = f1_score(y_test, y_pred, average='weighted')

      print(f"Accuracy: {accuracy:.4f}")
      print(f"Precision: {precision:.4f}")
      print(f"Recall: {recall:.4f}")
      print(f"F1 Score: {f1:.4f}")
```

```
Accuracy: 0.9837
Precision: 0.9898
Recall: 0.9837
F1 Score: 0.9867
```

```
[30]: # Access classifier and compute standard deviation
      rf_clf_step_over = rf_clf_over.named_steps['classifier']
      std_all = np.std([tree.feature_importances_ for tree in rf_clf_step_over.
        ↪estimators_], axis=0)

      # Get all feature names
      feature_names = rf_clf_over.named_steps['preprocessor'].get_feature_names_out()

      # Clean feature names by removing transformer prefixes
```

```
clean_names = [name.split("__")[-1] for name in feature_names]

# Create series with cleaned names
importances = pd.Series(rf_clf_step_over.feature_importances_,␣
↪index=clean_names)

# Get top 10 features
top10 = importances.sort_values(ascending=False).head(10).
↪sort_values(ascending=True)
top10_std = std_all[top10.index.map(clean_names.index)]

plt.figure(figsize=(10, 3))
ax = top10.plot.barh(xerr=top10_std)
ax.set_title("Random Forest: Top 10 Feature Importances")
plt.xlabel("Importance")
plt.tight_layout()
plt.show()
```



The plot trivially shows that `fico` is a paramount feature to include due to its high importance factor in our random forest model as we expected.

Comparing the baseline and final model's precision we have 0.9980 and 0.9898 respectively, the difference is negligable however our final model predicts a significantly narrower proportion as default. In comparison to our overly cautious baseline (which predicts a substatial amounts as FN's, which can be seen in our improved recall score) our final model is less so whilst retaining a similar precision without much negative impact. We prefer this this trade-off as while we would like to identify all individuals at risk of default, a bank cannot waste expenses and significant time to further examine so many individuals when the chance of most of the individuals defaulting is unlikely.

6.1 Testing model on active loans

Below we use our model to predict the active cases.

[31]:
```
# Predict and attach predictions to X_active
X_active = df_active.drop(['loan_status'], axis = 1)
```

```
X_active_with_preds = X_active.copy()
X_active_with_preds['prediction'] = rf_clf_over.predict(X_active)

# summary table with counts and proportions
prediction_summary = X_active_with_preds['prediction'].value_counts().
  ↪to_frame(name='count')
prediction_summary['proportion'] = (prediction_summary['count'] /␣
  ↪prediction_summary['count'].sum()).round(3)

print("Prediction summary:\n", prediction_summary)


X_pred_prepaid = X_active_with_preds[X_active_with_preds['prediction'] == 1]
X_pred_default = X_active_with_preds[X_active_with_preds['prediction'] == 0]
```

```
Prediction summary:
             count  proportion
prediction
1           71305       0.988
0             866       0.012
```

The results here for the active cases align with our expectations. The proportion of predicted default and prepaid cases closely reflects that of the distribution within the inactive cases. Indicating consistent model behaviour on unseen data.

7 Discussion & Conclusions

As discussed above our final model offers us a tradeoff between precision and recall, identifying geographic location, credit score, mortgage servicer and seller to be the most prevelant features in predicting loan defaults.

For future improvement, exploring location to a finer detail could be an improvement whilst taking into account the possibility of overfitting. Additionally, our model currently labels loans as default or prepaid, we could improve this further by categorising risk-of-default deeper than simply a binary value (e.g. low-risk, moderate-risk, high-risk).

We acknowledge the limitation that our final model is comprised almost solely of encoded categorical data which leads to an drastically increased feature count, potentially resulting in overfitting. However we still have significantly decreased the feature count from our baseline (which still provided fruitful results) while improving our recall score. Therefore we remain confident our final model is the best iteration to predict whether cases should be considered defaulted or prepaid.

8 Generative AI statement

Used for debugging, some plots too.

```
[ ]: # Run the following to render to PDF
     !jupyter nbconvert --to pdf ConsolidatedProject2.ipynb
```

[ ]:
```

```