project2

April 12, 2024

1 Machine Learning in Python - Project 2

Oliver Webb, Will Henshon, Ezra Muratoglu, James Zoryk

1.1 Setup

[1]:
```
import pandas as pd
import numpy as np
from scipy import stats
from scipy.stats import chi2_contingency
from math import sqrt
import warnings
import matplotlib.pyplot as plt
import seaborn as sns
plt.rcParams['figure.figsize'] = (8,5)
plt.rcParams['figure.dpi'] = 80
from warnings import simplefilter
import matplotlib

import sklearn
from sklearn.model_selection import train_test_split,GridSearchCV,␣
 ↪StratifiedKFold
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix,␣
 ↪make_scorer, fbeta_score,ConfusionMatrixDisplay,␣
 ↪accuracy_score,precision_score,recall_score,f1_score
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.exceptions import ConvergenceWarning
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.metrics import roc_auc_score
```

[2]:
```
fmac=pd.read_csv('freddiemac.csv')
```

2 Introduction

Managing risk plays a key role in financial markets. And a ubiquitous risk that most people are exposed to is their mortgage. Lenders take out significant risks loaning large sums of money in the form of mortgages, and managing this risk is vital given the large amounts of money being loaned as well as the secondary markets surrounding mortgage lending. When unchecked the concequences of these risks are evident in the 2008 financial crisis, the world-wide effects of which can still be seen many years later[1].

This report documents the process of building a classification model to predict whether a client will default. There exists some well established links between the client characteristics and their risk of defaulting; job stability, debt-to-income ratio, initial deposit and credit history among others all play a role in assessing the risk of a client defaulting[2]. Although measuring the impact of an individual client characteristic is straightforward, the interplay of different characteristics on the risk of defaulting is more nuanced. Here lies the motivation for the study; to produce an accurate model that is able to identify clients at a greater chance of defaulting in order to manage risk to lenders.

The model is built upon data coming from a dataset from the Federal Home Loan Mortgage Corporation, more often known as Freddie Mac[3]. The data comes from the Single Family Loan-Level Dataset, containing data on mortgages purchased or guaranteed by Freddie Mac between January 1, 1999 to 2023, although a subset of the full dataset will be used. The data includes standard information on the borrowings such as property information (location, size), features of the loan (maturity, fixed or variable rate), and information on the borrower (debt to income ratio, if they are a first time buyer). Importantly, the dataset is dominated by borrowers who have not defaulted on their loans.

3 Exploratory Data Analysis and Feature Engineering

To examine any invalid responses or entries in the dataset, tallys are made for each of the columns, to identify which features hold NaNs. This is further broken down into NaNs for defaulters and non-defaulters.

[3]:
```
'''
We desire to find out how many nans there are for which columns, and whether␣
 ↪those nans appear in the rows of defaulters or non-defaulters.
We identify columns with nans, initialize a dictionary, iterate through the␣
 ↪columns and then of those nans we tabulate how many are from defaulters /␣
 ↪non-defaulters
'''

## identify columns with nans
columns_with_nans = fmac.isna().sum()
columns_with_nans = columns_with_nans[columns_with_nans > 0].index.tolist()

## initialize dictionary to hold values
nans_info = {column: {'total_nans': 0, 'defaulters_nans': 0,␣
 ↪'non_defaulters_nans': 0} for column in columns_with_nans}
```

```
## iterate through columns
for column in columns_with_nans:

    ## total nans in the column, then tally nans for defaulters and non-defaulters
    total_nans = fmac[column].isna().sum()
    nans_info[column]['total_nans'] = total_nans
    defaulters_nans = fmac[column].isna() & (fmac['default'] == 1)
    non_defaulters_nans = fmac[column].isna() & (fmac['default'] == 0)

    nans_info[column]['defaulters_nans'] = defaulters_nans.sum()
    nans_info[column]['non_defaulters_nans'] = non_defaulters_nans.sum()

## convert to dataframe and display
nans_df = pd.DataFrame(nans_info).T
nans_df
```

[3]:

	total_nans	defaulters_nans	non_defaulters_nans
fico	1	1	0
cd_msa	594	24	570
ppmt_pnlty	38	0	38
flag_sc	5751	108	5643

In order to preserve the deafaulting datapoints, features such as 'flag_sc' (a flag associated with the loan exceeding Freddie Mac limits) - for which almost all of the defaulters have an invalid response - must be dropped. Removing such a large portion of the data would prove problematic when developing the model. 'cd_msa' (a code for the metropolitan area) is also dropped since these 15 defaulters make up over 20% of the defaulting data. Subsequently, there are no defaulting NaNs for 'ppmt_pnlty' (a flag noting whether the borrower has been obligated to pay a penalty) and only one for 'fico' so after dropping the two aforementioned features all NaNs are dropped.

At this point, two other features are also dropped: 'id_loan' and 'prepaid'. 'id_loan' gives a unique identifier to each loan. Given each identifier will be independent and unique to that loan, this is a feature that is not useful in prospectively predicting defaults. 'prepaid' indicates whether the loan has been paid off or not. This is a feature that is generated after defaults, given that defaulted loans have not been paid. Using this feature would result in a particularly accurate model, but it would presuppose the outcomes of loan defaults, which is information not available at the time of prediction.

These four features are therefore dropped, and thereafter all NaNs are dropped.

[4]:
```
## drop aforementioned columns, then drop all nans
df = fmac.drop(['cd_msa','flag_sc','id_loan','prepaid'], axis=1)
df = df.dropna()


## check for any other nans in whole dataframe
```

```
columns_with_nas = df.columns[df.isnull().any()].tolist()
print("Columns with NAs:", columns_with_nas)

## repeat of previous code to tally defaulters / non defaulters and total
df_total = df[['default']].groupby('default').size().reset_index(name='count')
total_count = df_total['count'].sum()
df_total.loc[len(df_total)] = ['Total', total_count]
print(df_total.to_string(index=False))
```

```
Columns with NAs: []
default  count
      0   5953
      1    112
  Total   6065
```

After dropping features that are oversaturated with NAs, we split our training and testing data.

[5]:
```
X1 = df.iloc[:, :-1]
X_train, X_test, y_train, y_test = train_test_split(X1, df.iloc[:,-1],␣
 ↪test_size=0.33, random_state=1, stratify=df.iloc[:,-1])
df = X_train
df['default']=y_train

df_test = X_test
df_test['default'] = y_test
```

[6]:
```
## calculate counts for defaulters and non-defaulters
df_total = df[['default']].groupby('default').size().reset_index(name='count')

# Calculate total count and append as new row
total_count = df_total['count'].sum()
df_total.loc[len(df_total)] = ['Total', total_count]

print(df_total.to_string(index=False))
```

```
default  count
      0   3988
      1     75
  Total   4063
```

Of the 3988 train entries, there are only 75 'defaulters'. The dataset is dominated by entries of clients who did not default. Considerable care will be required in order to ensure the data given to the model is representative of the unique features of the defaulters. Additionally, the model must be developed in such a way not to overfit to the non defaulting data and predict that no clients will default.

3.1 Visualizing Data and Detecting Significant Features

Numerical data will be analysed first. Boxplots are drawn up for all of these columns and shown for the defaulting and non defaulting clients. Features with different distributions between the two groups will be those that can be used in the model. Features with similar distributions suggest that there is no statsistical difference between the two, so these will not be useful in predicting whether a client is likely to default.

[7]:
```
'''
Here we want to display boxplots for all the numerical variables.
We define a list of the column names for the variables to be plotted, then␣
 ↪iterate through and plot
'''

variables = ['fico', 'cltv', 'dti', 'orig_upb', 'ltv', 'int_rt', 'mi_pct',␣
 ↪'orig_loan_term']
plt.figure(figsize=(20, 10))

## loop through the variables and create a subplot for each variable's boxplot
for i, variable in enumerate(variables, 1):
    plt.subplot(2, 4, i)
    sns.boxplot(x='default', y=variable, hue='default', data=df,␣
 ↪showfliers=False)
    plt.title(f'Boxplot of {variable}',size=22)
plt.tight_layout()
plt.show()
```

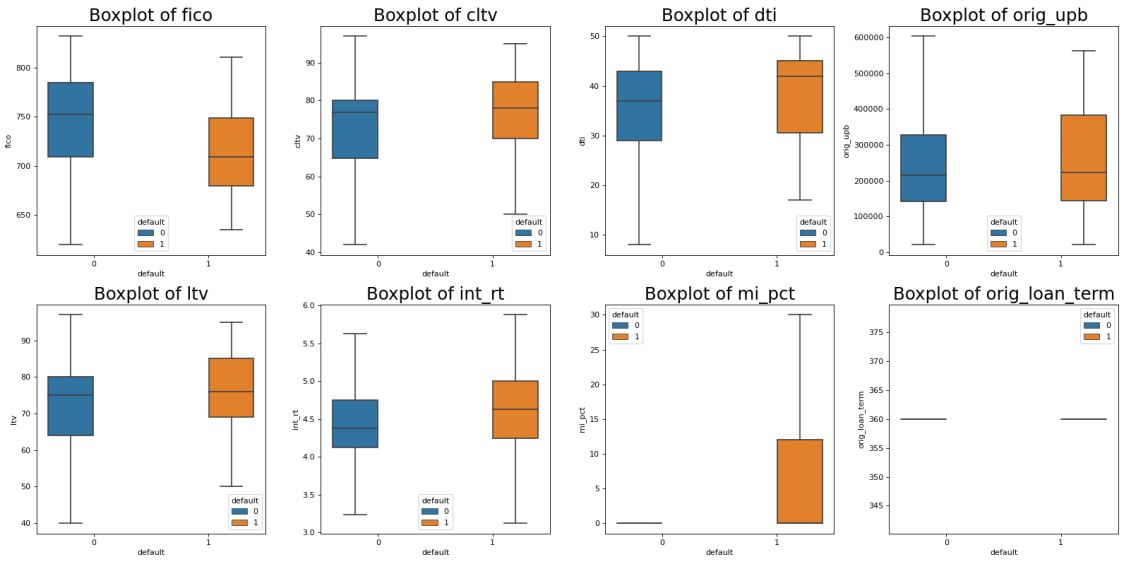

Fig. 1. Boxplots for numerical variables.

These plots give an impression of which features will be most useful to the model. Most of these

features show similar distributions between defaulters and non defaulters, although generally the variance is smaller on the plots for defaulters. This is to be expected given the smaller dataset, so the tails of these distributions will be cutoff sooner - moreso in these plots since all outliers have been included.

As a summary, fico is the credit score of the borrower, cltv is the original combined loan to value, dti is the original debt to income ratio for the borrower, orig_upb is the original UPB of the mortgage, ltv is the loan to value ratio, int_rt is the interest rate at the origin of the loan, and mi_pct is the mortgage insurance percentage (orig_loan_term is discussed below).

Plots for 'mi_pct' and 'orig_loan_term' look unusual, so these are examined in further detail as follows.

[8]:
```
## lists of features and plotting parameters
features = ['orig_loan_term', 'mi_pct', 'dt_matr', 'dt_first_pi']
default_status_labels = ['Non-Defaulters', 'Defaulters']
colors = ['blue', 'red']
fig, axes = plt.subplots(2, 4, figsize=(14, 6), sharex='col')

## iterate through features and plot histograms
for i, feature in enumerate(features):
    for j, default_status in enumerate([0, 1]):  ## 0 for Non-defaulters, 1 for␣
 ↪Defaulters
        ax = axes[j, i]
        sns.histplot(df[df['default'] == default_status][feature], ax=ax,␣
 ↪stat='probability', bins=20, color=colors[j], kde=False)
        ax.set_title(f'{default_status_labels[j]}: {feature}',size=16)
        ax.set_ylabel('Distribution',size=14)
        ax.tick_params(axis='y', rotation=50)

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

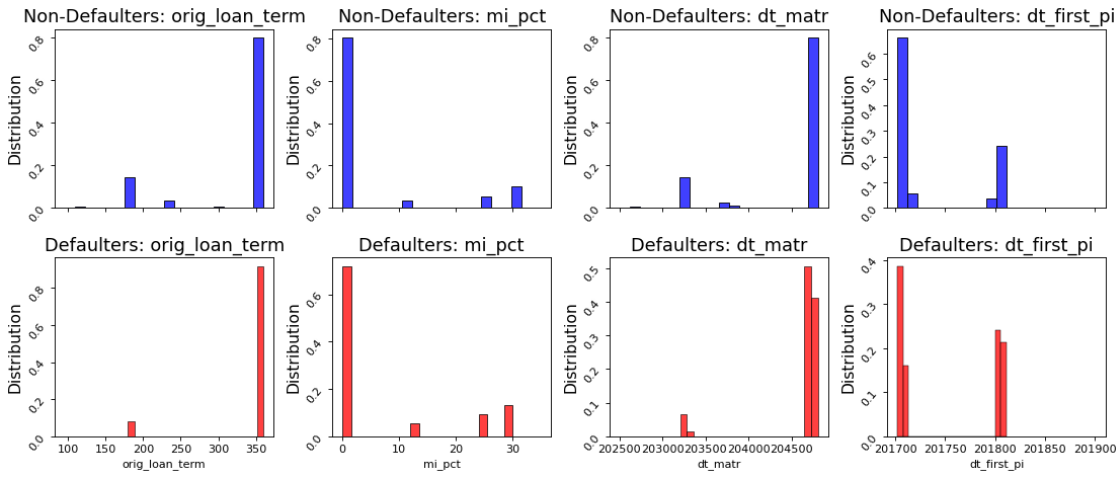

Fig. 2. Histograms for a selection of numerical variables.

For 'orig_loan_term' and 'mi_pct' the distributions are effectively the same for defaulters and non defaulters. The distributions are dominated by data centred on 360 and 0 respectively, which is what is seen in the boxplots above. Given how similar these distributions are between the two groups, there is no statistical significance and they are dropped from the dataframe.

The other numerical features can be given straight to the model. Although similar distributions are seen across the two groups, it does not significantly increase model complexity to leave these in, and if there is no corrolation between these features and the chance of a default, the weights for these features would be set to zero. It is also possible that theses features can be mainipulated to gain further insight into certain properties of the defaulters. For these reasons, the rest of the numerical features are left in at this stage.

Finally, orig_loan_term is a feature formed from dt_first_pi and dt_matr, which contain the dates of the first payment maturity of the mortgage respectively. Notwithstanding periods of uncertainty or financial volatility, a priori these features should have no bearing on the liklihood of a default; all kinds of clients will take out mortgages across a range of dates. To be sure there is no correlation, the distributions are plotted below. They show similar distributions between defaulters and non defaulters across both features. There are some differences in dt_first_pi, although this can be accounted for due to the small sample size for defaulters.

These four features can therefore be removed given the similarities between defaulters and non defaulters along with the understanding that dates should not impact individual clients - although the proportion of clients defaulting might increase in times of uncertainty.

[9]:
```
df = df.drop(['orig_loan_term','mi_pct','dt_matr','dt_first_pi'], axis=1)
df_test = df_test.drop(['orig_loan_term','mi_pct','dt_matr','dt_first_pi'],␣
↪axis=1)
```

3.1.1 Categorical Data

A similar analysis is performed for the categorical data. Contingency tables are used to visualize what proportion of the defaulters / non defaulters come from each category. Importantly, given the defaulters make up such a small fraction of the data - just under 2% - standard contingency tables would reflect this and show that the non-defaulters dominate in all categories. For this reason, a weighted contingency table is used; weights are calculated by the inverse of the group size, this will provide a sort of normalization. As an example, a feature where defaulters and non-defaulters are represented equally as a fraction of their group size would appear as a $0.5 : 0.5$ split in the table - whereas before it would have been $0.98 : 0.02$.

At risk of increasing complexity, it is also necessary to display what fraction of the data fits within a category of a certain feature. A feature may have 5 categories but have all data contained within one of those categories. Given splits between defaulters and non defaulters in one of the more sparsely populated categories would come with significant uncertainty due to the small sample size. This is discussed in more detail later on when analysing features with more than a few categories.

Before forming the tables, the number of unique categories for each feature is found. Features with only one unique category can be removed from the dataset given there are no differences between the defaulters and non defaulters here. Features with many unique categories are explored further on.

[10]:
```
'''
To find the number of unique entries (categories) for each feature.
Knowing the variables beforehand, we define a list of these column names and␣
 ↪use list comprehension to iterate through and tabulate the number of unique␣
 ↪entries for each column. Columns with only one unique entry are dropped
'''
categorical_feats = ['cnt_units', 'occpy_sts', 'channel', 'ppmt_pnlty',␣
 ↪'prop_type', 'loan_purpose', 'cnt_borr', 'flag_fthb', 'prod_type', 'st',␣
 ↪'seller_name', 'servicer_name','zipcode']

## dictionary to hold number of unique entries feature, then display
ents = {feat: df[feat].nunique() for feat in categorical_feats}
edf=pd.DataFrame(list(ents.items()),columns=['Feature','Unique Entries'])
print(edf.T.to_string(header=False))
```

```
Feature         cnt_units  occpy_sts  channel  ppmt_pnlty  prop_type
loan_purpose  cnt_borr  flag_fthb  prod_type  st  seller_name  servicer_name
zipcode
Unique Entries          4          3        3           1          5
3         2          3          1  53           29             32      652
```

[11]:
```
## drop these two features since they only have one unique entry
df = df.drop(['ppmt_pnlty', 'prod_type'],axis=1)
df_test = df_test.drop(['ppmt_pnlty', 'prod_type'],axis=1)
```

[12]:
```
'''
We now plot contingency tables for the categorical features.
To plot the tables we first separate the defaults and non defaults and␣
 ↪calculate the weights. A function is
used to plot a table given the feature being plotted.

A dictionary is then formed to store the percentages of 'how much of the data␣
 ↪fits into this category of the feature'.
Colourmaps and normalization factors are defined.
Subsequently the tables are formed as heatmaps, and the faces of the heatmap␣
 ↪are set to the percentage.
The tables show the split of data as the number, and the percentage as the␣
 ↪colour.
'''


## define features and calculate weights
feats = ['cnt_units', 'occpy_sts', 'channel', 'prop_type', 'loan_purpose',␣
 ↪'cnt_borr', 'flag_fthb']
df['weight']=df['default'].apply(lambda x:1/df[df['default']==x].shape[0])

## calculate weighted crosstab
def weighted_crosstab(df, x, y, weight):
```

```
    return pd.crosstab(df[x],df[y],df[weight],aggfunc='sum',normalize='index')

## calculate breakdown tables and maximum percentage
breakdown_tables = {}
max_percentage = 0
for feat in feats:
    count_breakdown = df.groupby('default')[feat].value_counts().
 ↪unstack(fill_value=0)
    count_breakdown = count_breakdown.astype(int)
    total_counts = count_breakdown.sum(axis=0)
    total_percentage = (total_counts / df.shape[0] * 100).round(1)
    count_breakdown.loc['Total %'] = total_percentage
    breakdown_tables[feat] = count_breakdown
    max_percentage = max(max_percentage, total_percentage.max())

## create colormap and normalization
cmap = plt.cm.Blues
norm = plt.Normalize(vmin=0, vmax=max_percentage)
annot_kws = {'size': 14, 'weight': 'bold', 'color': '#DAA520'}

## plotting
warnings.filterwarnings('ignore')
fig, axes = plt.subplots(2, 4, figsize=(24, 14))
axes = axes.flatten()

## map categories to their percentages
c_pct={feat: breakdown_tables[feat].loc['Total %'] for feat in feats}

## plotting
for i, feature in enumerate(feats):
    ct = weighted_crosstab(df, feature, 'default', 'weight')
    sns.heatmap(ct, ax=axes[i], annot=True, fmt=".2f", cbar=False, cmap=cmap,␣
 ↪annot_kws=annot_kws)

    for y in range(len(ct.index)):
        for x in range(len(ct.columns)):
            category = ct.index[y]
            color_val = c_pct[feature].get(category, 0)
            color = cmap(norm(color_val))
            axes[i].add_patch(plt.Rectangle((x, y), 1, 1, fill=True,␣
 ↪color=color, edgecolor='grey'))

    axes[i].set_title(f"Weighted Contingency Table: \n{feature} vs Default",␣
 ↪size=22)
    axes[i].set_ylabel(f'{feature}', size=16)
    axes[i].set_xlabel('Default', size=16)
```

```
## colorbar, then hide the last subplot
sm = plt.cm.ScalarMappable(cmap=cmap, norm=norm)
sm.set_array([])
cbar_ax = fig.add_axes([0.1, 0.05, 0.8, 0.03])
cbar = fig.colorbar(sm, cax=cbar_ax, orientation='horizontal')
cbar.set_label('Percentage of Total Data', size=24)
axes[7].set_visible(False)
fig.tight_layout(rect=[0, 0.11, 1, 1])
plt.show()
```

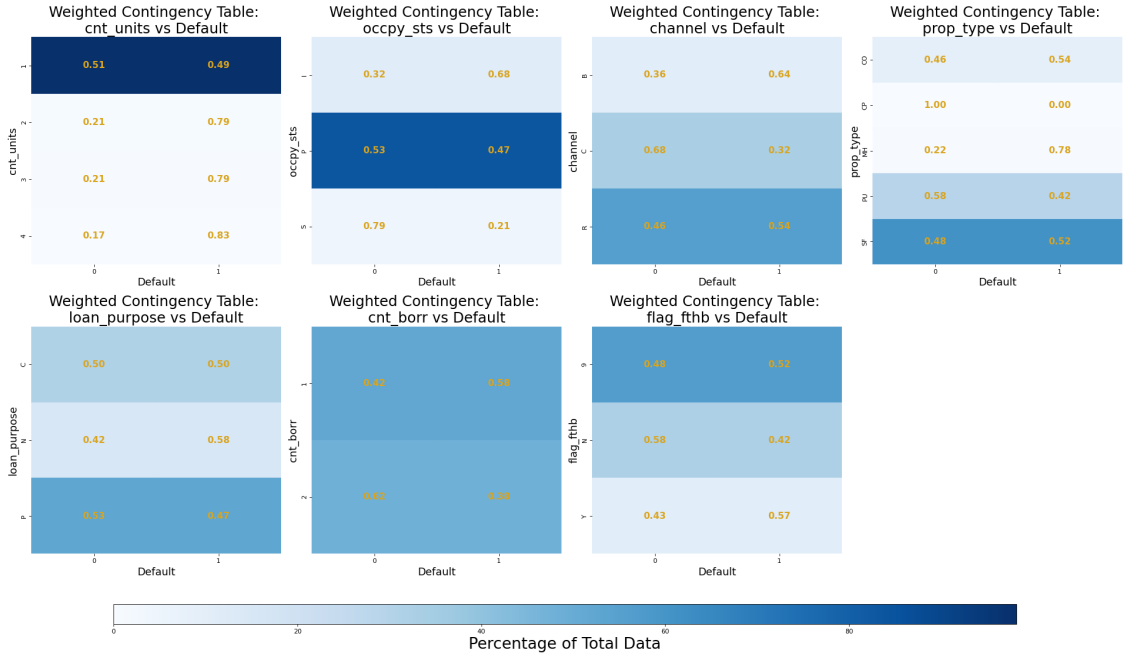

Fig. 3. Weighted contingency tables, coloured by percentage of data, for catagorical features.

To guide the reader, the numbers within each category of the table show how well represented defaulters and non defaulters are for that category - 0.5 : 0.5 split tells us that both defaulters and non defaulters are represented in proportion to their sample size. The colours on the tables tell us what fraction of the *total* dataset falls into that category for that feature.

To interpret the tables, it is important to know the categories for each predictor. cnt_units is the number of units in the property; denotes the property type, where P means owner-occupied, I means investment property, and S means second home; channel says whether a broker or correspondent was involved with the loan, with B indicating broker, C indicating correspondent, and R indicating retail. prop_type is for property type, and has 5 categories: CO for condominium, CP for cooperative share, MH for manufactured home, SF for single-family home, and PU for planned unit development. Loan purpose can be cash out refinance (C), no cash-out refinance (N), unspecified (R), or purchase mortgage (P). cnt_borr is the number of people on the mortgage, and flag fthb refers to a first-time homebuyer, where 9 is unspecified.

As an example, for prop_type, the final category (SF) forms most of the data and clients under

this category are evenly split between defaulters and non defaulters according to their sample size. There is a significant split for CP: all clients with CP for prop_type are non defaulters, although given the colour (white) very few clients have CP for prop_type to begin with.

Looking only at the split between defaulters and non defaulters, one would judge prop_type, occpy_sts and cnt_units to be the most useful features to the model given these have the largest split (all over 70% for certain categories). However, the categories that face the largest split happen to be those that have the smallest sample size; the small sample size introduces uncertainty and using these features can bias the model towards outcomes that might not be true to the test data. It is therefore essential to balance the split of defaulters in a category to the total sample size for that category.

With this in mind, cnt_borr appears to be a good feature. There is both a notable split and a balance in the sample size. Although occpy_sts appears to have its data centred around one category, the split for the other two is significant so this is included also. The penultimate category for prop_type (PU) shows a significant split and a resonable sample size. Due to the small sample sizes across most features strategies can be adopted to encode only some categories from a given feature.

[13]:
```
df = df.drop(['cnt_units','weight'], axis=1)
df_test = df_test.drop(['cnt_units'], axis=1)
```

Unexplored features so far include 'zipcode' , 'seller_name' , 'servicer_name' , and 'st' (state). These are all categorical features, although as seen before, they contain many more categories than the other features. To examine these, bar plots are drawn up showing the number of occurances of each category within its feature - for example, how many times IL appears in 'state' - for both defaulters and non defaulters, the following frequency ratio is then computed.

$$\text{frequency ratio} = \frac{\text{defaulters proportion}}{\text{non-defaulters proportion}} = \frac{\text{category defaulters}}{\text{total defaulters}} \div \frac{\text{category non defaulters}}{\text{total non defaulters}}$$

Where category defaulters is the number of defaulters that had that particular category for the relevant feature - 5 defaulters had IL for 'st' feature. category non defaulters is the equivalent for non defaulters. This ratio allows consideration of whether the number of defaulters that appear for a certain category is representative of their frequencies in the whole dataset. If the ratio is 1, then the two proportions are equal, and defaulters proportionaly represent their share of that category; this would be expected for a variable entirely independent to defaulting. A ratio > 1 would suggest that this category 'attracts' defaulters, and those with this category for the relevant feature are more likely to default - for example, clients with IL for state feature, are more likely to default. The converse is true for a ratio < 1.

Lastly, the category names in 'seller_name', 'servicer_name' and 'zipcode' (zipcode contains integers but these can be considered to be categories) are encoded for graphical brevity. These can be decoded afterwards to find the names corresponding to the numbers. It is confirmed that the number of unique entries is constant. Plots are analysed briefly individually, then summarized together.

[14]:
```
'''
Here we want to replace the entries of the 3 features with numbers
```

```
This is done for clarity and so the axes can be read. We setup dictionaries to␣
 ↪store the maps, the create new columns in the dataframes. These columns are␣
 ↪deleted after plotting
'''

encoding_maps = {}
decoding_maps = {}

for column in ['seller_name', 'servicer_name', 'zipcode']:
    encoding_maps[column] = {k: v for v, k in enumerate(df[column].unique())}
    df[f'{column}_encoded'] = df[column].map(encoding_maps[column])          ␣
 ↪## apply the encoding
    decoding_maps[column] = {v: k for k, v in encoding_maps[column].items()} ␣
 ↪## create reverse mapping
```

[15]:
```
## here we check that the encoded columns have the same number of unique␣
 ↪entries as the original ones
print(f"len(seller_name)==len(seller_name_encoded):     ␣
 ↪{df['seller_name_encoded'].nunique()==df['seller_name'].nunique()}")
print(f"len(servicer_name)==len(servicer_name_encoded): ␣
 ↪{df['servicer_name_encoded'].nunique()==df['servicer_name'].nunique()}")
print(f"len(zipcode)==len(zipcode_encoded):             {df['zipcode_encoded'].
 ↪nunique()==df['zipcode'].nunique()}")
```

```
len(seller_name)==len(seller_name_encoded):      True
len(servicer_name)==len(servicer_name_encoded):  True
len(zipcode)==len(zipcode_encoded):              True
```

[16]:
```
def decode(encoded_values, column):
    '''
    Now columns have been encoded, we need to decode them
    This function allows us to input a number for a given feature, and map it␣
 ↪back to the original category name. The function uses the decoding map and␣
 ↪returns the name(s) of the original category(s)
    '''

    ## if we input a single number
    if isinstance(encoded_values, int):
        return decoding_maps[column][encoded_values]
    ## for a list of numbers
    elif isinstance(encoded_values, list):
        return [decoding_maps[column][num] for num in encoded_values]
    else:
        raise ValueError("Input must be an integer or a list of integers.")
```

[17]:
```
## we define these functions to simplify and standardize plotting

## function to plot one barplot for a given column
def plot_frequency(df, column, frequency, order, title, ylabel, ax):
    sns.barplot(x=frequency.index, y=frequency.values, order=order,␣
 ↪palette="viridis", ax=ax)
    ax.set_title(title,size=18)
    ax.set_xlabel(column,size=16)
    ax.set_ylabel(ylabel,size=16)
    ax.tick_params(axis='x', rotation=70)

def plot_state_frequency(df, column, top, order_by_proportion=False):
    '''
    Function to plot tallies for a given category:
    number of non-defaulters, number of defaulters, and the proportion ratio.
    '''
    warnings.simplefilter(action='ignore', category=FutureWarning)

    ## calculate frequencies and proportions
    freq_counts = df.groupby(['default', column]).size().unstack(fill_value=0)
    total_counts = df['default'].value_counts()
    proportions = freq_counts.div(total_counts, axis='index')
    ratio = proportions.loc[1] / proportions.loc[0]

    ## sort by proportion or non-default frequency
    if order_by_proportion:
        order = ratio.sort_values(ascending=False).iloc[:top].index
    else:
        order = freq_counts.loc[0].sort_values(ascending=False).iloc[:top].index

    ## hardcode titles and y labels given we know what is being plotted
    titles = ['Frequency for Non-defaulters', 'Frequency for Defaulters',␣
 ↪'Defaulters to Non-defaulters Proportion Ratios']
    ylabels = ['Frequency', 'Frequency', 'Proportion Ratio']

    fig, axes = plt.subplots(3, 1, figsize=(14, 9))

    ## iterate through and plot
    for i, (freq, title, ylabel) in enumerate(zip([freq_counts.loc[0],␣
 ↪freq_counts.loc[1], ratio], titles, ylabels)):
        plot_frequency(df, column, freq.reindex(order), order, f'{column}␣
 ↪{title}', ylabel, axes[i])
        if i == 2:
            axes[i].axhline(1, color='red', linestyle='--')
    plt.tight_layout()
    plt.show()
```

[18]:
```
plot_state_frequency(df, 'zipcode_encoded',top=105,order_by_proportion=True)
```

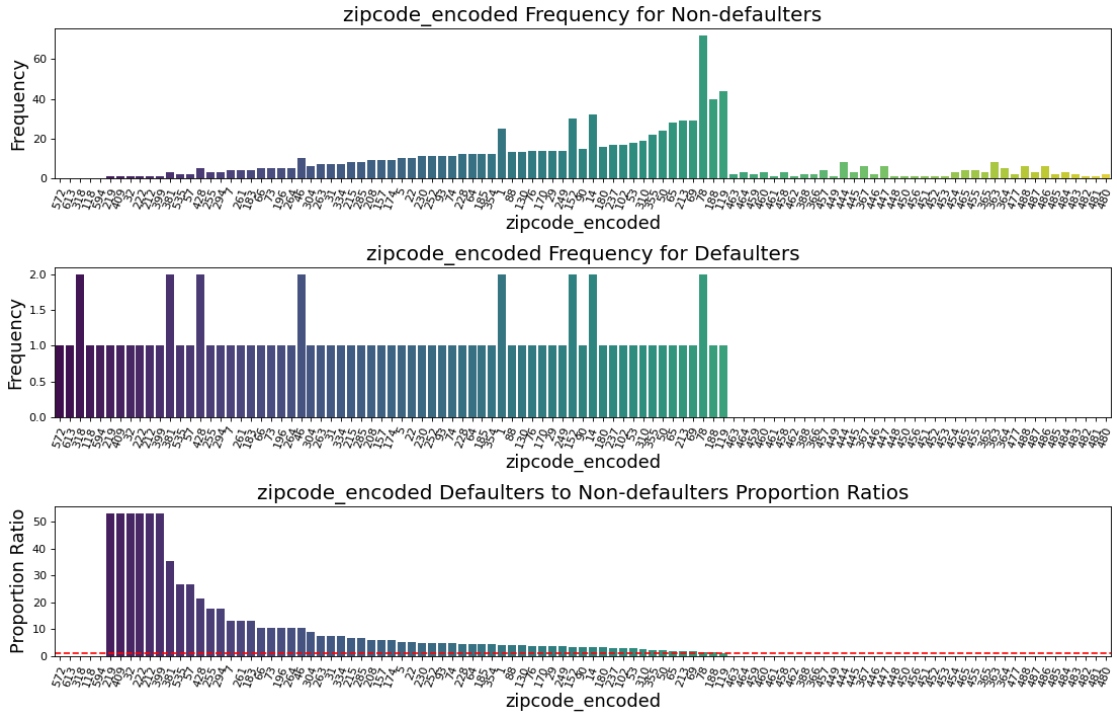

Fig. 4. Barplots for zipcode feature.

There are 717 unique entries in the zipcode feature and with roughly 100 defaulters, the vast majority of these zipcodes have no defaulters. From the second plot, no zipcode has more than two defaulters. Given the small defaulters sample size and the large number of zipcodes, the data is exceedingly sparse. Ordering by our proportion ratio, we see that zipcodes with an overrepresentation of defaulters primarily come from zipcodes with a small number of non-defaulters; meaning that the ratio is high because of a small sample size, not because of an influx of defaulters. This is confirmed by the trend seen in the first strip, ordering by ratio descending also happens to (approximately) order by non-defaulting frequency ascending.

[19]:
```
plot_state_frequency(df, 'st',top=df['st'].nunique(),order_by_proportion=False)
```

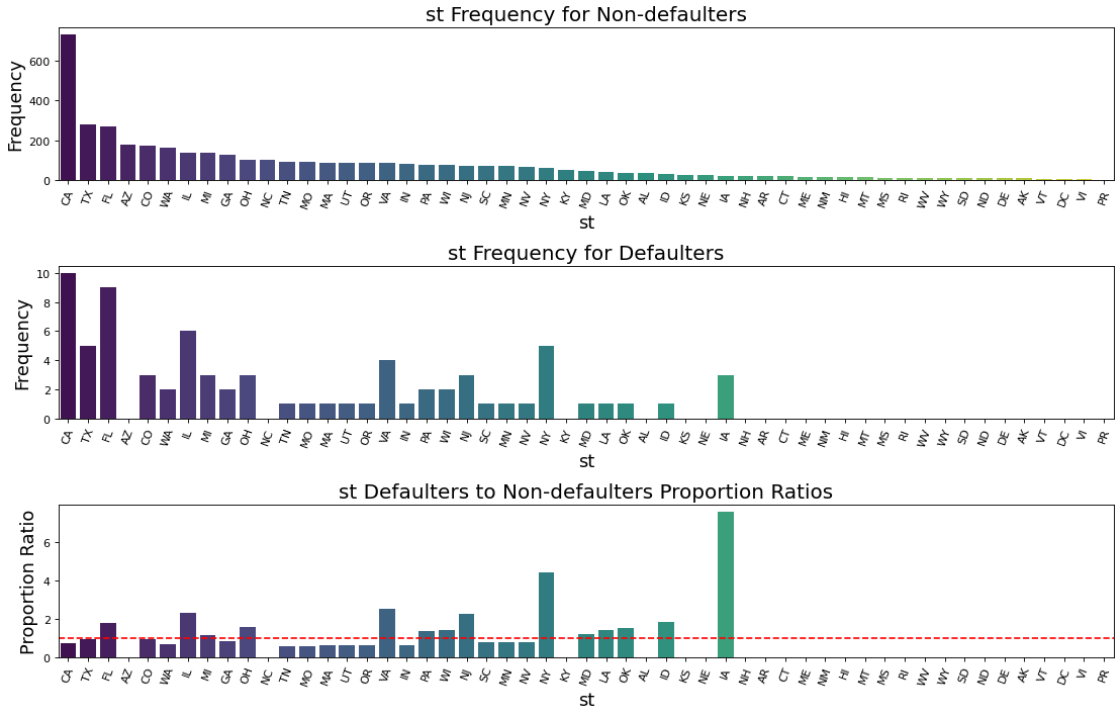

Fig. 5. Barplots for state 'st' feature.

With 53 unique entries, and a certain number of states without any defaulters a distribution is seen with a notable variance in the number of defaulters from each state. Defaulters are well represented in certain states such as CA, TX, CO among others, with proportion ratios around 1. However defaulters are over and underrepresented in a number of states. AZ serves as a good example, with no defaulters coming from this state - given a client who comes from AZ, it is unlikely that they are a defaulter.

Encoding all states would introduce uneccessary complexity into the model. Weights for states with a proportion ratio of 1 would be set to zero, since defaulters are equally represented. To make use of this split in certain states, the most skewed states can be encoded. It is wise to note that further down the plots, the sample size becomes smaller; so although CT has a large proportion ratio, this comes with a large uncertainty given the overall sample of clients from CT is small.

[20]:
```
plot_state_frequency(df,'seller_name_encoded',top=df['seller_name_encoded'].
↪nunique())
```

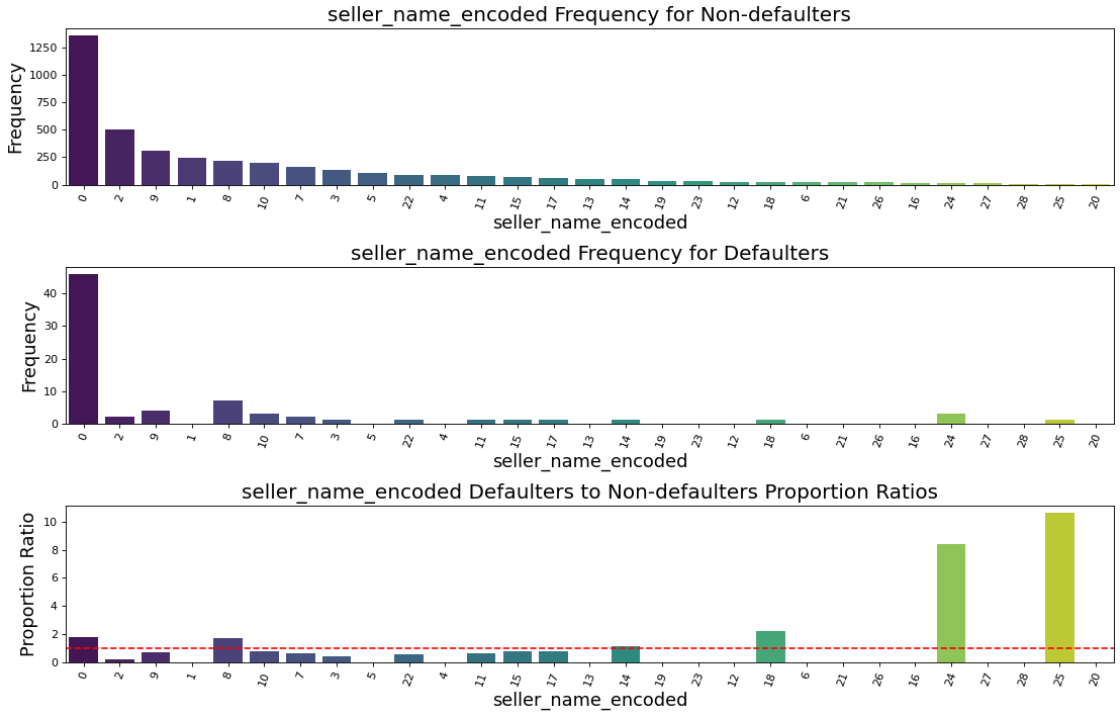

Fig. 6. Barplots for 'seller_name' feature.

The distribution seen for seller_name shows a mostly even proprtion ratios with exceptions for smaller total sample sizes for a given category (24 and 25). The data of defaulters is largely dominated by 0 (corresponding to 'other sellers'), leading to a slight overrepresentation of defaulters. Out of the 112 defaulters, approximatley 45 are in the 0 category, leading all other categories to be sparsely populated. For real-world use, including categories such as 24 and 25 would introduce great uncertainty and variability into the model. In some cases, test data may match the distribution seen here, leading to great accuracy. But otherwise, it would be plausible that the model could be biased towards always predicting default for a client coming from 24 or 25.

With this in mind, sellers towards the left of the plots (with larger sample sizes) should be favoured.

[21]:
```
plot_state_frequency(df, 'servicer_name_encoded',top=df['servicer_name'].
↪nunique())
```

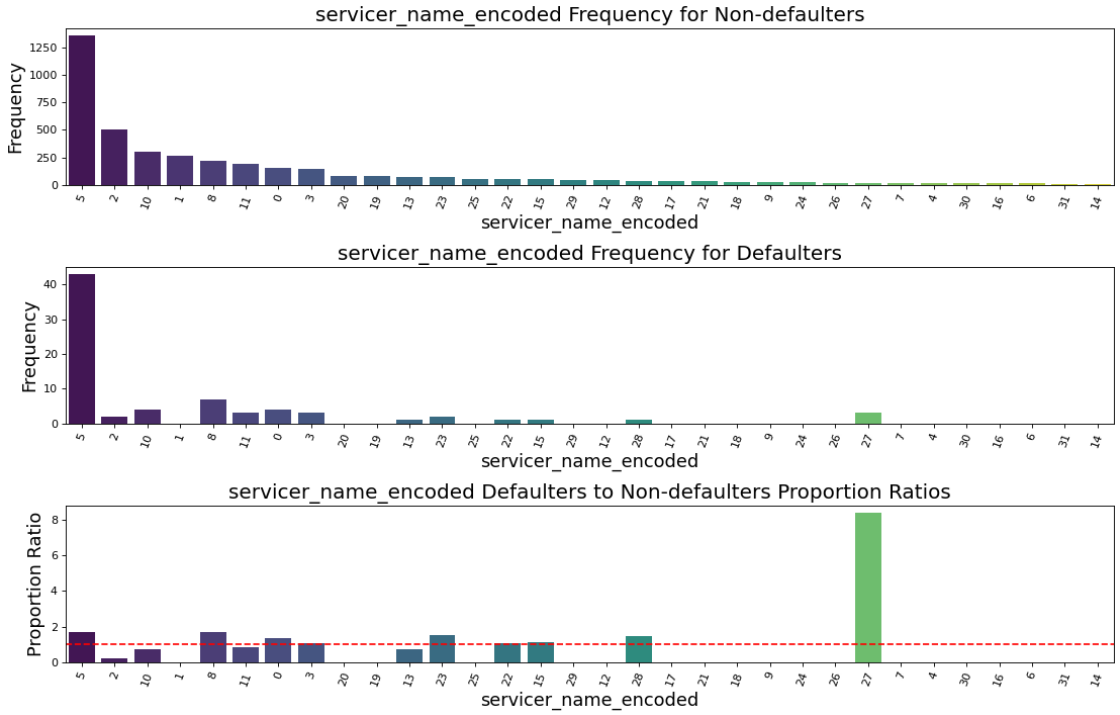

Fig. 7. Barplots for 'servicer_name' feature.

As with seller_name, most defaulters fall into the 0 category (corresponding to other servicers). Following the same procedure, categories with a large total sample size are favoured over those with a smaller one.

Summarizing the four plots, it has been discussed how categories with a small total sample size (a small number of defaulters and non defaulters fit into this category) come with significant uncertainty. This is likely to introduce bias to the model in the cases where the test or train data does not reflect the distribution seen here. Therefore, choices of which categories to include are centred around which categories are well represented but still show an over or underrepresentation of defaulters. A category with an even representation of defaulters (proportion ratio equal to one), would have little impact on model performance, since such features would not provide a way of distinguishing between defaulters or non defaulters.

Considering this, and in order to maintain model simplicity, categories are chosen based on their sample size and their proportion ratio; where large sample sizes are favoured along with proportion ratios furthest from one. This is done in a qualitative manner, although given the simple nature of the dataset at hand, it is straightforward to distinguish these categories.

Regarding the zipcode data, and its small sample size for all zipcodes, the choice is made to include the four best represented zipcodes with small proprtion ratios. This is confirmed by setting order_by_proprtion=False, in order to disregard those categories where the small sample size leads to a very large (>50) proportion ratio and set the y-axis to a more workable scale.

The below code indicates the choices of categories made, the decoding process and subsequent encoding using OneHotEncoder.

[22]:
```
df = df.
 ↪drop(['seller_name_encoded','servicer_name_encoded','zipcode_encoded'],axis=1)
```

[23]:
```
def encode_specific_category(df, column, category_name):
    """
    To better understand the steps, and to encode one category of a feature we␣
 ↪implement this function.
    We input the dataframe, the column / feature and the category within that␣
 ↪feature
    """
    ## use lambda function to set entries of encoded column to 1 if entries in␣
 ↪feature column match category name
    df[f'{column}_{category_name}'] = df[column].apply(lambda x: 1 if x ==␣
 ↪category_name else 0)
    return df
```

[24]:
```
'''
We have decided upon the categories in the following dictionary.
They are decoded to get the name, and then encoded into their own column
The dictionary is formed to easily iterate through, and an if statement is used␣
 ↪to check for the 'st' feature
The previously defined function is used to encode a given category of a feature
Finally we can drop the original columns
'''

features_to_encode = {
    'zipcode': [23,44,71,85,57,28,148,112,34,80,203,7],
    'st': ['FL','AZ','IL','NC','TN','IN','MA','VA','NY','IA'],
    'seller_name': [0,13,15,6,17,16,21,25,27],
    'servicer_name': [0,15,8,17,9,12,10,20,5,1,24]}

## iterate over dictionary
for feature, categories in features_to_encode.items():
    for category in categories:

        ## check if intiger
        if isinstance(category, int):
            ## decode to find original name
            cat_name = decode(category, feature)
        else:
            cat_name = category  ## f the category is a string, use it as is

        ## encode the specific category
        df = encode_specific_category(df, feature, cat_name)
        df_test = encode_specific_category(df_test, feature, cat_name)

## drop the original columns after encoding
```

```
df = df.drop(columns=features_to_encode.keys())
df_test = df_test.drop(columns=features_to_encode.keys())
```

[25]:
```
'''
We also desire to encode the previously analysed features
We use a similar process as earlier, encoding certain categories of a feature
'''

## initialize dictionary
categories_to_encode = {
    'occpy_sts': ['I', 'S'],
    'channel': ['B', 'C', 'R'], ## all categories
    'prop_type': ['PU'],
    'loan_purpose': ['N'],
    'cnt_borr': ['1', '2'],  ## all categories
    'flag_fthb': ['N', 'Y']}

## iterate over the dictionary and encode the specific categories
for column, categories in categories_to_encode.items():
    for category in categories:
        df = encode_specific_category(df,column,category)
        df_test = encode_specific_category(df_test,column,category)

## drop the original columns after encoding
df = df.drop(columns=categories_to_encode.keys())
df_test = df_test.drop(columns=categories_to_encode.keys())
```

4 Model Fitting and Tuning

[26]:
```
def make_confusion_matrices(y_actual_train, y_predict_train, title_train,
                            y_actual_test, y_predict_test, title_test):
    '''
    Generates a confusion matrix for a given model
    '''
    fig, ax = plt.subplots(1, 2, figsize=(10, 5))

    cm_train = confusion_matrix(y_actual_train, y_predict_train, labels=[0, 1])
    disp_train = ConfusionMatrixDisplay(confusion_matrix=cm_train,
                                        display_labels=["No", "Yes"])
    disp_train.plot(cmap='Greens', colorbar=True, ax=ax[0])
    ax[0].set_title(title_train,size=18)
    ax[0].tick_params(axis=u'both', which=u'both', length=0)
    ax[0].grid(visible=False)

    # Confusion matrix for test data
    cm_test = confusion_matrix(y_actual_test, y_predict_test, labels=[0, 1])
```

```
    disp_test = ConfusionMatrixDisplay(confusion_matrix=cm_test,
                                       display_labels=["No", "Yes"])
    disp_test.plot(cmap='Greens', colorbar=True, ax=ax[1])
    ax[1].set_title(title_test,size=18)
    ax[1].tick_params(axis=u'both', which=u'both', length=0)
    ax[1].grid(visible=False)

    plt.tight_layout()
    plt.show()

def␣
 ↪get_metrics_score(model,X_train_df,X_test_df,y_train_pass,y_test_pass,statsklearn,threshold
 ↪5,flag=True,roc=False):
    '''
    Function to calculate different metric scores of the model - Accuracy,␣
 ↪Recall, Precision, and F1 score
    model: classifier to predict values of X
    X_train_df, X_test_df: Independent features
    y_train_pass,y_test_pass: Dependent variable
    statsklearn : 0 if calling for Sklearn model else 1
    threshold: thresold for classifiying the observation as 1
    flag: If the flag is set to True then only the print statements showing␣
 ↪different will be displayed. The default value is set to True.
    roc: If the roc is set to True then only roc score will be displayed. The␣
 ↪default value is set to False.
    '''
    # defining an empty list to store train and test results

    score_list=[]
    if statsklearn==0:
        pred_train = model.predict(X_train_df)
        pred_test = model.predict(X_test_df)
    else:
        pred_train = (model.predict(X_train_df)>threshold)
        pred_test = (model.predict(X_test_df)>threshold)


    pred_train = np.round(pred_train)
    pred_test = np.round(pred_test)

    train_acc = accuracy_score(y_train_pass,pred_train)
    test_acc = accuracy_score(y_test_pass,pred_test)

    train_recall = recall_score(y_train_pass,pred_train)
    test_recall = recall_score(y_test_pass,pred_test)

    train_precision = precision_score(y_train_pass,pred_train)
```

```
    test_precision = precision_score(y_test_pass,pred_test)

    train_f1 = f1_score(y_train_pass,pred_train)
    test_f1 = f1_score(y_test_pass,pred_test)


    score_list.
↪extend((train_acc,test_acc,train_recall,test_recall,train_precision,test_precision,train_f1

    if flag == True:
        print("\x1b[0;30;47m \033[1mMODEL PERFORMANCE\x1b[0m")
        print("\x1b[0;30;47m \033[1mAccuracy    : Train:\x1b[0m",
              round(accuracy_score(y_train_pass,pred_train),3),
              "\x1b[0;30;47m \033[1mTest:\x1b[0m ",
              round(accuracy_score(y_test_pass,pred_test),3))
        print("\x1b[0;30;47m \033[1mRecall      : Train:\x1b[0m"
              ,round(recall_score(y_train_pass,pred_train),3),
              "\x1b[0;30;47m \033[1mTest:\x1b[0m" ,
              round(recall_score(y_test_pass,pred_test),3))

        print("\x1b[0;30;47m \033[1mPrecision   : Train:\x1b[0m",
              round(precision_score(y_train_pass,pred_train),3),
              "\x1b[0;30;47m \033[1mTest:\x1b[0m ",
              round(precision_score(y_test_pass,pred_test),3))
        print("\x1b[0;30;47m \033[1mF1          : Train:\x1b[0m",
              round(f1_score(y_train_pass,pred_train),3),
              "\x1b[0;30;47m \033[1mTest:\x1b[0m",
              round(f1_score(y_test_pass,pred_test),3))
        make_confusion_matrices(y_train_pass, pred_train,"Confusion Matrix for␣
↪Train",y_test_pass, pred_test, "Confusion Matrix for Test")

    if roc == True:

        print("\x1b[0;30;47m \033[1mROC-AUC Score  :Train:\x1b[0m: ",
              round(roc_auc_score(y_train_pass,pred_train),3),
              "\x1b[0;30;47m \033[1mTest:\x1b[0m: ",
              round(roc_auc_score(y_test_pass,pred_test),3))

    return score_list # returning the list with train and test scores
```

To generate a baseline model, we simply take our data and run a logistic regression. This form of regression (as opposed to linear regression) is necessary here because we are predicting a binary variable in terms of a probability. We also weigh the defaulted observations more than the non-defaulted ones: as discussed above, there are vastly more non-defaulted observations, so we would likely end up in a situation where the model predicts too many people not to default. This would introduce a large amount of liability in this situation, since the most costly result here is that we predict a mortgage not to default, and it ends up defaulting.

[27]:
```
#Define the X and Y columns
feature_columns = df.columns.drop('default')
X_train = df[feature_columns]
y_train = df['default']
X_test = df_test[feature_columns]
y_test = df_test['default']

numerical_transformer = Pipeline(steps=[
    ('scaler', StandardScaler())
])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, feature_columns)
    ])

lr = LogisticRegression(solver='newton-cg', random_state=1,␣
 ↪fit_intercept=False, class_weight={0: 0.15, 1: 0.85})

pipeline_main_v1 = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', lr)
])

# Fit and valuate the model
fit = pipeline_main_v1.fit(X_train, y_train)
statmodel = 0
scores_pipeline_main = get_metrics_score(pipeline_main_v1, X_train, X_test,␣
 ↪y_train, y_test, statmodel)
```

```
MODEL PERFORMANCE
Accuracy   : Train: 0.539  Test:  0.545
Recall     : Train: 0.893  Test: 0.865
Precision  : Train: 0.035  Test:  0.034
F1         : Train: 0.067  Test: 0.066
```

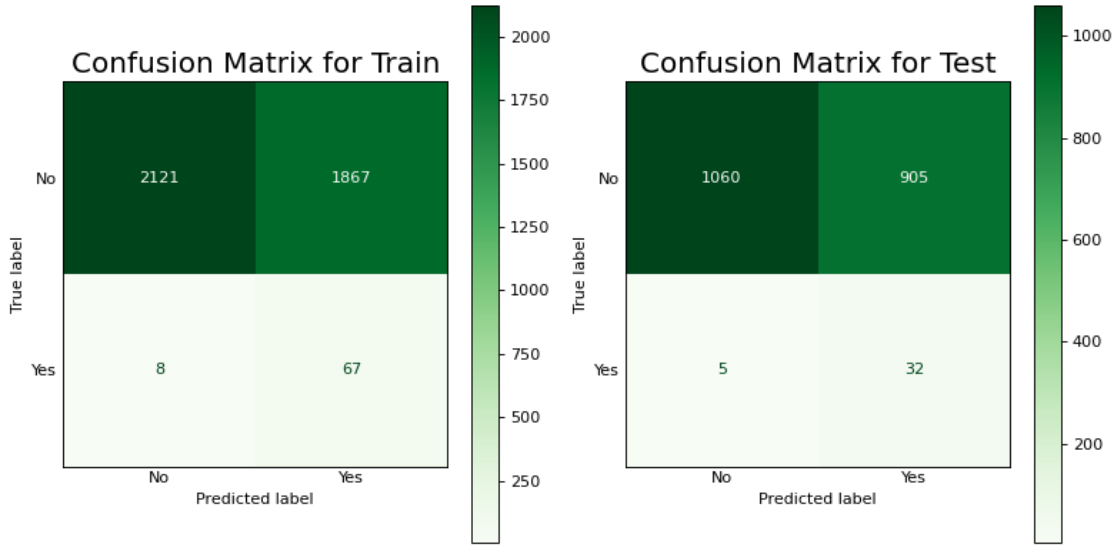

Fig. 8. Confusion matrices for the first model.

After fitting the model, we then produce confusion matrices for the training and testing data, along with summary statistics. We see that the accuracy of the model is about 54% for the training case and 55% for the test case, which is relatively low. However, we only have 8 cases in the training data, and 5 cases in the test data, in which we predict a non-default and there is actually a default. As stated above, this is the quantity that we most want to minimize, and this model does a reasonable job of minimization here.

The recall score is a measure of this: this is a measurement that compares how many people were predicted to default with how many people actually defaulted. This score is .893 for the train set and .865 for the test set. The model thus does a permissable job of "covering" the defaulters; however, we would like to see a recall closer to 1, as every mislabeled defaulter is a liability to the bank.

The precision and F1 score are less encouraging for this model. The precision measures the ratio of falsely predicted non-defaults to the full amount of non-defaults. This score is very low, meaning that we predict far too many positives in order to "cover" the true positives. The F1 score, which reflects a combination of the precision and recall, also reflects this. However, though these numbers look bad, this is not necessarily too much of a problem. The model inaccurately predicting defaults means that it recommends taking on less risk, which is acceptable.

4.0.1 Variance Inflation Factor (VIF)

Multicollinearity arises when the predictor variables being considered for the regression model are highly correlated among themselves. This can lead to unstable and unreliable estimates of the regression coefficients. A widely accepted method for detecting multicollinearity is the use of Variance Inflation Factors (VIF).[4]

VIF measures how much the variances of the estimated regression coefficients are inflated compared to when the predictor variables are not linearly related. The largest VIF value among all predictor

variables is often used as an indicator of the severity of multicollinearity. A maximum VIF value exceeding 10 is frequently taken as an indication that multicollinearity may be unduly influencing the least squares estimates.[5]

It is important to note that while VIF is a useful diagnostic tool for detecting multicollinearity, it has a limitation in that it cannot distinguish between several simultaneous multicollinearities. Therefore, it is essential to combine the VIF analysis with domain knowledge and understanding of the relationships between the predictor variables. To calculate the VIF for each feature, we use the following code:

[28]:
```
# Calculate the VIF for each feature
vif = pd.DataFrame()
vif["Feature"] = X_test.columns
vif["VIF"] = [variance_inflation_factor(X_test.values, i) for i in range(X_test.
  ↪shape[1])]
```

[29]:
```
high_vif_features = vif[vif['VIF'] > 12]['Feature'].tolist()

print(high_vif_features)
# Remove the high VIF features from the dataframe
X_test = X_test.drop(columns=high_vif_features)
```

```
['cltv', 'ltv', 'seller_name_USAAFEDSAVINGSBANK',
'servicer_name_USAAFEDSAVINGSBANK', 'channel_B', 'channel_C', 'channel_R']
```

'cltv' and 'ltv': These features are likely to be highly correlated because they both measure the ratio of the loan amount to the property value.

'seller_name_WELLSFARGOBANK,NA', 'seller_name_BANKOFAMERICA,NA', 'seller_name_GUILDMTGECO' and 'servicer_name_WELLSFARGOBANK,NA', 'servicer_name_BANKOFAMERICA,NA', 'servicer_name_GUILDMTGECO': Here we see that the seller_name and servicer_name are the same in these cases.

'channel_B', 'channel_C', 'channel_R':The correlation could be due to certain sellers having a preference for specific loan types, borrower profiles, or geographic areas.

4.0.2 Hyperparameter Tuning using GridSearchCV

To further improve the performance of our logistic regression model, we performed hyperparameter tuning using Grid Search Cross Validation. We defined a parameter grid with various combinations of regularization strength, penalty type (L1 or L2), solver (how Python computes the model), and class weights (how much the model weighs defaulters vs. non-defaulters). Then, we used cross validation (the process of dividing the data into sections, or folds, and then holding out one fold at a time from the fitting process to avoid overfitting) to find the set of parameters with the best model performance. Since as discussed above, recall is the most important value for evaluating this model, we use this as the metric to find the best possible model.

[ ]:
```
warnings.filterwarnings("ignore", category=ConvergenceWarning)
```

```
feature_columns = [col for col in feature_columns if col not in␣
 ↪high_vif_features]
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, feature_columns)
    ])

lr = LogisticRegression(solver='newton-cg', random_state=1, fit_intercept=False)

pipeline_main_v1 = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', lr)
])

# Define the hyperparameter grid for GridSearchCV
param_grid = {
    'classifier__C': [0.01, 0.1, 1, 10, 100],
    'classifier__penalty': ['l1', 'l2'],
    'classifier__solver': ['liblinear', 'saga', 'lbfgs'],
    'classifier__class_weight': [None, 'balanced',{0: 0.15, 1: 0.85}, {0: 1, 1:␣
 ↪5}, {0: 1, 1: 10}]
}

# Create the GridSearchCV object
grid_search = GridSearchCV(pipeline_main_v1, param_grid, cv=5,␣
 ↪scoring='recall', n_jobs=-1)

# Fit the GridSearchCV object on the training data
grid_search.fit(X_train, y_train)

# Print the best hyperparameters and score
print("Best hyperparameters: ", grid_search.best_params_)
print("Best score: ", grid_search.best_score_)

# Evaluate the model with the best hyperparameters
best_model = grid_search.best_estimator_
statmodel = 0  # 0 for sklearn and 1 for statmodel
scores_best_model = get_metrics_score(best_model, X_train, X_test, y_train,␣
 ↪y_test, statmodel)
```

Fig. 9. Confusion matrices for the second model.

After running the grid search, we see that for minimizing recall, the best possible set of parameters are as follows: 0.01 for the regularization strength, indicating that the model prefers small weights for parameters as opposed to prioritizing lager weights; balanced class weights, indicating that we do not want to weigh the observations differently; and an L1 classifier penalty, which suggests that the model prefers a lasso-style approach, where many of the parameter weights are set to 0, as opposed to a ridge approach, where they are more balanced.

Here, we see that the overall accuracy of the model drops from 54% to 52% in in the train case goes to 54% in the test case. However, the recall of this model is much better; we now only have 9 false predictions of default in the training case and still 5 in the test case. This means that this model performs slightly worse than the first version of the model.

Therefore, since it is simpler, and performs slightly better, we select the first model for the purpose of recommending mortgage purchases. It protects the bank reasonably well against purchasing mortgages that will turn out to default. It should be noted that we tried various other forms of classification, including random forest-based methods; however, these were found to be less effective than the cross-validated logistic one.

[ ]:
```
# Retrieve the best logistic regression estimator
best_lr_model = grid_search.best_estimator_.named_steps['classifier']

# Get the coefficients from the logistic regression model
coefficients = best_lr_model.coef_[0]

# Retrieve the feature names which were processed by ColumnTransformer
feature_names = np.array(feature_columns)

# Create a DataFrame for easier visualization
importance_df = pd.DataFrame({
    'Feature': feature_names,
    'Weight': coefficients
})

# Sort the features by the absolute value of their coefficient
importance_df = importance_df.reindex(importance_df.Weight.abs().
 ↪sort_values(ascending=False).index)

# Filter out the top 25 features
importance_df_filtered = importance_df[:25]

# Plot the feature importance
plt.figure(figsize=(15, 8))
plt.barh(importance_df_filtered['Feature'], importance_df_filtered['Weight'])
plt.xlabel('Feature Weight', size=18)
plt.ylabel('Feature', size=18)
plt.title('Feature Importance from Logistic Regression', size=20)
plt.gca().invert_yaxis()  # To display the most important feature at the top
plt.show()
```

Fig. 10. Feature importance in the second model.

We can now evaluate the significant features in this model so that we can deduce the most important factors for predicting default likelihood. First, we see that fico score is the most important factor. This has a negative value, so increasing fico score is associated with less likelihood of default, which fits our intuition. Next, we see that the mortgage being from a seller outside of the list of the most common ones is correlated to a higher likelihood of defaulting. It may be possible that these

smaller banks have lower standards for mortgages than bigger ones, or simply serve areas with higher default likelihood.

Higher original loan to value ratio correlates with defaults: this means that buyers who make smaller down payments are more likely to default, which seems intuitively likely. Higher interest rate at the start of the loan also is correlated with higher default rates. This also makes intuitive sense: these loans are inherently more difficult to pay due to being higher interest, and the bank giving the initial mortgage likely only gives lower rates to customers who it deems more likely to pay the loan back. We also see that investment properties are more likely to default than other types when accounting for the other factors.

By observing the graph, we can see several other high-importance factors. These include the location of the mortgage by state: Iowa has higher rates of default, and we also see that various servicers and sellers are significantly more or less likely to have their mortgages default. These are not easily summarized, but are useful information for our decision making process in the cases where they apply.

5 Discussion & Conclusions

In developing the classification model, the largest challenge was the small sample of defaulters in the dataset. Many of the features showed an even representation of defaulters or non defaulters in proportion to their sample size; meaning that they could not be used to distinguish between the two groups given the information in the feature. Although some features or categories therein did show splits between the two groups (suggesting, for example, that it may more likely a client defaults if they are from a certain state) upon further analysis it was found that some of the features showing the largest splits had an inbalance in how well represented the two groups were. The prop_type feature illustrates this well; of clients with 'MH' for this categorical feature, 92% of them were defaulters. Although less than 5% of prop_type entries had 'MH', corresponding to an extremley small number of defaulters. These features had to be ruled out, as including them could introduce bias and uncertainty into the model.

The features that were ultimately included generally tended to be features favouring non defaulters: for example, states with a large number of non defaulters and a small number of defaulters. These features, when present, tell us that it is unlikely that the client will default. There are a number of features and categories that favoured defaulters: certain states, servicers or sellers of loans, along with numerical featurs such as credit score all presented as increased liklihood of defaulting. For the servicers and sellers, this was chiefly 'other servicers / sellers'; it is possible that the named banks (such as JP Morgan, Wells Fargo, Bank of America) are more rigorous in their analysis of mortgage applications, whereas smaller banks may take on more risk given the smaller sums being dealt with, leading them to sell to clients with a higher risk of defaulting.

None of the numerical feature appeared to have particularly different distributions between defaulters or non defaulters. The distributions for defaulters tended to have a smaller variance, likely due to the smaller sample size. However slight trends could be seen in some features: 'fico' indicating credit score, did show that the median credit score of a non defaulter was higher than that of a defaulter, although there were still cases of clients with high credit scores defaulting, and those with low credit scores in the non defaulter category. Representation within a category does not apply for these numerical features - all clients have data for them - so aside from those with particularly similar distributions ('orig_loan_term' and 'mi_pct'), all features were left in. It does

not significantly increase model complexity, or introduce bias into the model; weights will be set to zero for features unable to distinguish between the two groups.

In terms of conclusions, we see that the most valuable generally applicable information for predicting the likelihood of default are fico score and original loan to value ratio along with original loan interest rate: higher fico score correlates with less likelihood of defaults, and vice versa for the interest rate and original loan to value ratio. Additionally, we saw that many categorical features were valuable for predicting default likelihood: among these were property type, location, seller, and more.

These results generally agree with the literature. In [4], the authors take a Bayesian approach, and focus more fully on the numerical predictors instead of putting emphasis on categorical predictors as we did. Still, among their results were the conclusion that for both fico score and original loan interest rate, there is a statistically significant difference in the posterior mean between the default case and the prepayment case. This means that the results support our claim that these are significant predictors for default likelihood (they do not include original loan to value ratio).

As a reminder from the modeling section, the accuracy of this model is not generally high. It is not especially effective at predicting when people are unlikely to default. However, since non-defaulters are an overwhelming majority of the dataset, this is still enough that we recommend many mortgages for purchase. More importantly, the model is optimized to minimize the mortgages that are predicted to be paid but still default. These are the costliest outcomes, so we want to avoid them at almost all costs. The selected model is relatively good at avoiding these, although without additional data on how much the bank will make from mortgages and lose from defaults, it is difficult to analyze whether it is good enough.

Finally, it should be noted that this dataset is an oversimplification of real-life data. While it is a good starting point, it only includes prepaid and defaulted loans from about 6 to 8 years ago; most loans from this time period are still active. This means our dataset excludes most of the data, so we should remain skeptical about blindly applying the model to new data.

6 References

[1] Campello, Graham, Harvey, *The Real Effects of Financial Constraints: Evidence from a Financial Crisis* 2009, doi: 10.3386/w15552

[2] Elul et Al. *What "Triggers" Mortgage Default?* 2010 American Economic Review vol. 100,2

[3] Freddie Mac, *Single Family Loan Dataset*, https://www.freddiemac.com/research/datasets/sf-loanlevel-dataset

[4] Bhattacharya, Wilson, Soyer, *A Bayesian approach to modeling mortgage default and prepayment* 2018, doi: https://doi.org/10.1016/j.ejor.2018.10.047

[5] Kutner, M. H., Nachtsheim, C. J., Neter, J., & Li, W. (2005). Applied linear statistical models (5th ed.). McGraw-Hill/Irwin.

[6] Weisberg, S. (2005). Applied linear regression (3rd ed.). John Wiley & Sons.

[ ]:
```
# Run the following to render to PDF
!jupyter nbconvert --to pdf project2.ipynb
```

[ ]: