Project2

April 10, 2024

1 Machine Learning in Python - Project 2

Due Friday, April 12th by 4 pm.

Axel Eichelmann, Darragh Ferguson, Heather Napthine, Will Spence

[4]:
```
# Data libraries
import pandas as pd
import numpy as np

# Plotting libraries
import matplotlib.pyplot as plt
import seaborn as sns

# Set plotting defaults
plt.rcParams['figure.figsize'] = (8, 5)
plt.rcParams['figure.dpi'] = 80

# sklearn modules for data preprocessing
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder,␣
 ↪PolynomialFeatures

# sklearn modules for model selection and evaluation
from sklearn.model_selection import train_test_split, GridSearchCV,␣
 ↪RandomizedSearchCV
from sklearn.model_selection import KFold, StratifiedKFold
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.metrics import accuracy_score, roc_auc_score,␣
 ↪precision_recall_curve
from sklearn.metrics import mean_squared_error, make_scorer, recall_score,␣
 ↪precision_score

# sklearn and imblearn modules for model training
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC, LinearSVC
from sklearn.calibration import CalibratedClassifierCV
from sklearn.pipeline import Pipeline, make_pipeline
```

```
from imblearn.pipeline import Pipeline as ImPipeline
from imblearn.over_sampling import RandomOverSampler

# Utility modules
import warnings
import joblib
from time import time
from collections import Counter
from scipy.stats import uniform, loguniform

# sklearn datasets for example data
from sklearn.datasets import make_classification, make_moons

# Prevent warnings
warnings.filterwarnings('ignore')
```

[5]:
```
# Load data in easyshare.csv
df = pd.read_csv("freddiemac.csv")
```

2 Introduction

The ability to predict mortgage defaults is critical to the success of a financial lending institution for multiple reasons such as assessing risk levels, facilitating regulatory compliance and improving inverstor confidence, all of which ultimately improve the company's profitability. When making these predictions there are various factors which need to be considered, and as such the development of such prediction models can become an extremely complicated matter during which interpretability of the model is often lost.

In this report we present a model which predicts whether or not an individual will default given various factors such as FICO scores, duration of loans, interest rate of the loan and many other variables. In particular this model was developed using the data provided in a simplified version of the 'Single Family Loan-Level Dataset' provided by Freddie Mac [1], and makes use of the logistic regression modelling technique. On top of this we performed a detailed investigation to evaluate the importance (coefficients) assigned by the model to each feature when making these predictions.

In doing so we are not only providing a tool which can be used to set optimal terms for the loan by choosing the best interest rates, setting appropriate loan terms, etc., thus minimizing financial losses, but we are also developing the knowledge base of Freddie Mac employees that read this report so that they can make indivualised decisions where necessary, rather than blindly following the guidance set forth by the model in all scenarios.

3 Exploratory Data Analysis and Feature Engineering

Before beginning the data exploration process we divided the overall dataset into training and testing groups, in order to avoid data leakage. We chose the testing group to consist of 30% of the entries in the overall dataset, and fixed a random before performing this splitting in order facilitate the reproducibility of these results.

On top of this we stratified the splitting of the data by the 'default' column to ensure that there was equal proportions of default cases in the training and testing datasets.

[6]:
```
# Fix a random seed for reproducible results.
seed = np.random.seed(42)
# Immediately split dataframe into test and training sets so as to avoid data␣
 ↪leakage.
df_train, df_test = train_test_split(df, test_size = 0.3, stratify =␣
 ↪df['default'], random_state = seed)
```

After performing this splitting we used various functions to get an overall picture of the training dataset. In particular we looked at certain important factors such as the count of NA entries, the types of data that each feature consisted of (e.g. categorical or numeric), the distribution of values for the features, etc.

Another important thing to note is that we calculated the proportion of default cases within the training dataset to identify if there was any imbalancing issues with the data. Indeed, as was expected, we found that the proportion of default cases was significantly less than the proportion of non-default cases, thus this data was indeed imbalanced.

[7]:
```
# Describe the data
print(f"Overall there are {df_train.isna().sum().sum()} NA values in the␣
 ↪training dataset, excluding the identified values from the documentation.")
# Get an overview of the training data using the '.info()' and '.describe()'␣
 ↪functions
df_train.info()  # This show that the majority of NA values come from the␣
 ↪'flag_sc' and 'cd_msa' columns
df_train.describe().round(2)  # We use this to get an idea of the distribution␣
 ↪of data points for each feature
# We want to see the proportion of defaults in the dataset
default_cases = df_train[df_train.default == 1]
default_proportion = df_train[df_train.default == 1].shape[0]/df_train.shape[0]
print(f"Overall, the training dataset contains {default_cases.shape[0]} \
default cases out of a total of {df_train.shape[0]} observations.")
print(f"This means that only {round(default_proportion*100,2)}% of the dataset␣
 ↪were default cases \
thus we need to account for this imbalanced data when fitting our␣
 ↪classification model")
```

```
Overall there are 4480 NA values in the training dataset, excluding the
identified values from the documentation.
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4272 entries, 4891 to 2790
Data columns (total 28 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   fico            4271 non-null   float64
 1   dt_first_pi     4272 non-null   int64
```

```
 2   flag_fthb       4272 non-null   object
 3   dt_matr         4272 non-null   int64
 4   cd_msa          3859 non-null   float64
 5   mi_pct          4272 non-null   int64
 6   cnt_units       4272 non-null   int64
 7   occpy_sts       4272 non-null   object
 8   cltv            4272 non-null   int64
 9   dti             4272 non-null   int64
 10  orig_upb        4272 non-null   int64
 11  ltv             4272 non-null   int64
 12  int_rt          4272 non-null   float64
 13  channel         4272 non-null   object
 14  ppmt_pnlty      4240 non-null   object
 15  prod_type       4272 non-null   object
 16  st              4272 non-null   object
 17  prop_type       4272 non-null   object
 18  zipcode         4272 non-null   int64
 19  id_loan         4272 non-null   object
 20  loan_purpose    4272 non-null   object
 21  orig_loan_term  4272 non-null   int64
 22  cnt_borr        4272 non-null   int64
 23  seller_name     4272 non-null   object
 24  servicer_name   4272 non-null   object
 25  flag_sc         238 non-null    object
 26  prepaid         4272 non-null   int64
 27  default         4272 non-null   int64
dtypes: float64(3), int64(13), object(12)
memory usage: 967.9+ KB
Overall, the training dataset contains 79 default cases out of a total of 4272
observations.
This means that only 1.85% of the dataset were default cases thus we need to
account for this imbalanced data when fitting our classification model
```

On top of this general investigation into each of the features, we looked at the column 'loan_id' which was supposed to have a unique value for each row entry in the dataset since they all corresponded to a unique mortgage. After confirming that this was indeed the case we dropped this column as it could not be used to predict whether a mortgage was likely to be defaulted or not.

After this initial exploration of the data we looked into the 'flag_sc' and 'cd_msa' variables to determine there were any reasons behind the significant amounts of missing values for these features.

The description of the features in the original data source says that 'flag_sc' identifies 'mortgages that exceed conforming loan limits with origination dates on or after 10/1/2008 and were delivered to Freddie Mac on or after 1/1/2009.' Upon looking at the unique values in this column we saw that the entries are either NaN or 'Y'. Due to the binary nature of this feature we assumed that the NaN values were representative of cases where the mortgage was not flagged for this reason, thus we did not remove them when cleaning the data.

Similarly, the description of the 'cd_msa' variable which describes the Metropolitan Statistical Area where the property is located specifies that null entries indicate cases where 'the area in which the

mortgaged property is located is a) neither an MSA nor a Metropolitan Division, or b) unknown', thus null entries represent one of the categories for this variable and so we did not remove the null values for this feature from the clean dataset either.

[8]:
```
# Create copy of df_train and df_test which will contain the cleaned up version
 ↪of the data.
df_train_clean = df_train.copy()
df_test_clean = df_test.copy()
# Check that all 'id_loan' values are unique.
print(f"Out of the {df_train_clean.shape[0]} entries in the dataset, there are
 ↪{df_train_clean.id_loan.nunique()} unique 'id_loan' entries,\
 thus this we can confirm that each entry corresponds to a different mortgage
 ↪and drop the 'id_loan' column.")
# Drop the column.
df_train_clean = df_train_clean.drop(columns=['id_loan'])
df_test_clean = df_test_clean.drop(columns=['id_loan'])
# Investigation into the 'flag_sc' variable
df_train_clean[df_train_clean.flag_sc.notnull()]['flag_sc'].unique()
df_test_clean[df_test_clean.flag_sc.notnull()]['flag_sc'].unique()
# Investigation into the 'cd_msa' variable
df_train_clean[df_train_clean.flag_sc.notnull()]['flag_sc'].unique()
df_test_clean[df_test_clean.flag_sc.notnull()]['flag_sc'].unique()
```

```
Out of the 4272 entries in the dataset, there are 4272 unique 'id_loan' entries,
thus this we can confirm that each entry corresponds to a different mortgage and
drop the 'id_loan' column.
```

[8]:
```
array(['Y'], dtype=object)
```

The documentation describes multiple instances of the inclusion of values such as 9 or 99 that actually represent data that was 'Not Available'. In order to determine whether or not to remove these cases from the dataset we counted the number of instances of such entries for each feature. In doing so we found that the 'cltv', 'ltv' and 'dti' features each had one such case, the 'flag_fthb' feature had 2416 such cases, and the rest of the features did not have any of these cases.

Due to the very low number of such instances for the 'cltv', 'ltv' and 'dti' features we decided to remove these entries from the data. In the case of the 'flag_fthb' feature we decided against performing this same removal of data since it would dramatically reduce the overall amount of mortgage cases data with which we could fit a prediction model. On top of this we noted that a reading of '9' here simply meant that the borrower did not fall neatly into the categories of "Yes" or "No" for this question, this does not necessarily render the borrower useless for modelling so we simply kept '9' as a third category for this variable.

[9]:
```
# Calculate counts of 'not available' cases for each feature
fico_count = (df_train_clean['fico'] == 9999.0).sum()
mi_pct_count = (df_train_clean['mi_pct'] == 999).sum()
cltv_count = (df_train_clean['cltv'] == 999).sum()
dti_count = (df_train_clean['dti'] == 999).sum()
```

```
cnt_units_count = (df_train_clean['cnt_units'] == 99).sum()
prop_type_count = (df_train_clean['prop_type'] == "99").sum()
ltv_count = (df_train_clean['ltv'] == 999).sum()
flag_fthb_count = (df_train_clean['flag_fthb'] == "9").sum()
channel_count = (df_train_clean['channel'] == "9").sum()
occpy_sts_count = (df_train_clean['occpy_sts'] == "9").sum()
loan_purpose_count = (df_train_clean['loan_purpose'] == "9").sum()
# Create a DataFrame with the counts
counts_df = pd.DataFrame({
    'Variable': ['fico', 'mi_pct', 'cltv', 'dti', 'cnt_units', 'prop_type',␣
 ↪'ltv', 'flag_fthb', 'channel', 'occpy_sts', 'loan_purpose'],
    'Count': [fico_count, mi_pct_count, cltv_count, dti_count, cnt_units_count,␣
 ↪prop_type_count, ltv_count, flag_fthb_count, channel_count, occpy_sts_count,␣
 ↪loan_purpose_count]
})
print(counts_df)
# Apply filtering conditions for numeric values where '999' is used to indicate␣
 ↪missing data.
df_train_clean = df_train_clean[(df_train_clean['cltv'] != 999) &␣
 ↪(df_train_clean['dti'] != 999) & (df_train_clean['ltv'] != 999)]

df_test_clean = df_test_clean[(df_test_clean['cltv'] != 999) &␣
 ↪(df_test_clean['dti'] != 999) & (df_test_clean['ltv'] != 999)]
```

```
        Variable  Count
0           fico      0
1         mi_pct      0
2           cltv      1
3            dti      1
4      cnt_units      0
5      prop_type      0
6            ltv      1
7      flag_fthb   2416
8        channel      0
9      occpy_sts      0
10  loan_purpose      0
```

Looking at the entries in the 'ppmt_pnlty' column we saw that they were either 'N' or null. Since the description of this feature does not mention the existence of null entries, we were unable to confirm whether these null entries should be either 'Y' or 'N'. This combined with the fact that there were only 32 cases of such null entries led us to decide to remove them from the clean dataset. We also removed the single row in the dataset which correspondeded to the NaN entry for the 'fico' feature for the same reasons.

Further investigation into the 'ppmt_pnlty' feature showed that all remaining non-NaN values for this feature showed that they all took on the same value of '1' meaning that this feature was redundant for making prediction about defaults, thus we dropped this entire feature from the dataset.

The same was the case for the 'prod_type' feature which described whether a mortgage was a fixed-rate mortgage or an adjustable-rate mortgage. Indeed we saw that every entry for this feature showed the value 'FRM' thus every mortgage in the dataset corresponded to a fixed-rate mortgage, and so we dropped this feature from the overall dataset.

The final column which we decided to drop was the 'prepaid' one. This time we saw that its values were always the complement of the values of the 'default' column thus this 'prepaid' column alone could be used to determine whether a mortgage was defaulted or not. As a result including this feature in the model building process would have only allowed for an artifically perfect model, providing no idication as to what features serve as good predictors of mortgage defaults.

[10]:
```
# Drop rows with NaN entries in either 'ppmt_pnlty' or 'fico'
df_train_clean.fico.isna().sum()
df_train_clean.ppmt_pnlty.isna().sum()
df_train_clean = df_train_clean.dropna(subset=['fico','ppmt_pnlty'])
df_test_clean = df_test_clean.dropna(subset=['fico','ppmt_pnlty'])

# Check the unqiue values in the 'ppmt_pnlty' feature
df_train_clean['ppmt_pnlty'].unique()   # output = 1
print(f"There is only {df_train_clean['ppmt_pnlty'].nunique()} unqiue value for␣
 ↪the 'ppmt_pnlty' thus we drop this feature")
df_train_clean = df_train_clean.drop(columns=['ppmt_pnlty'])
df_test_clean = df_test_clean.drop(columns=['ppmt_pnlty'])

# Check the unique values for the 'prod_type' feature
print(f"Out of the {df_train_clean.shape[0]} entries in the dataset, there are␣
 ↪{df_train_clean.prod_type.nunique()} unique 'prod_type' entries,\
 thus we drop the 'prod_type' column.")
df_train_clean = df_train_clean.drop(columns=['prod_type'])
df_test_clean = df_test_clean.drop(columns=['prod_type'])

# Check if 'prepaid' equals 1 minus 'default' for all rows.
if (df_train_clean['prepaid'] == 1 - df_train_clean['default']).all():
    print("All 'prepaid' values are the complement of 'default' so we drop this␣
 ↪column.")
else:
    print("Not all 'prepaid' values are the complement of 'default'.")

# Drop the column.
df_train_clean = df_train_clean.drop(columns=['prepaid'])
df_test_clean = df_test_clean.drop(columns=['prepaid'])
```

```
There is only 1 unqiue value for the 'ppmt_pnlty' thus we drop this feature
Out of the 4237 entries in the dataset, there are 1 unique 'prod_type' entries,
thus we drop the 'prod_type' column.
All 'prepaid' values are the complement of 'default' so we drop this column.
```

After making these changes to the data, we wanted to get an idea of how the remaining features were related to each other. To do this we examined their relationships with the 'default' feature to

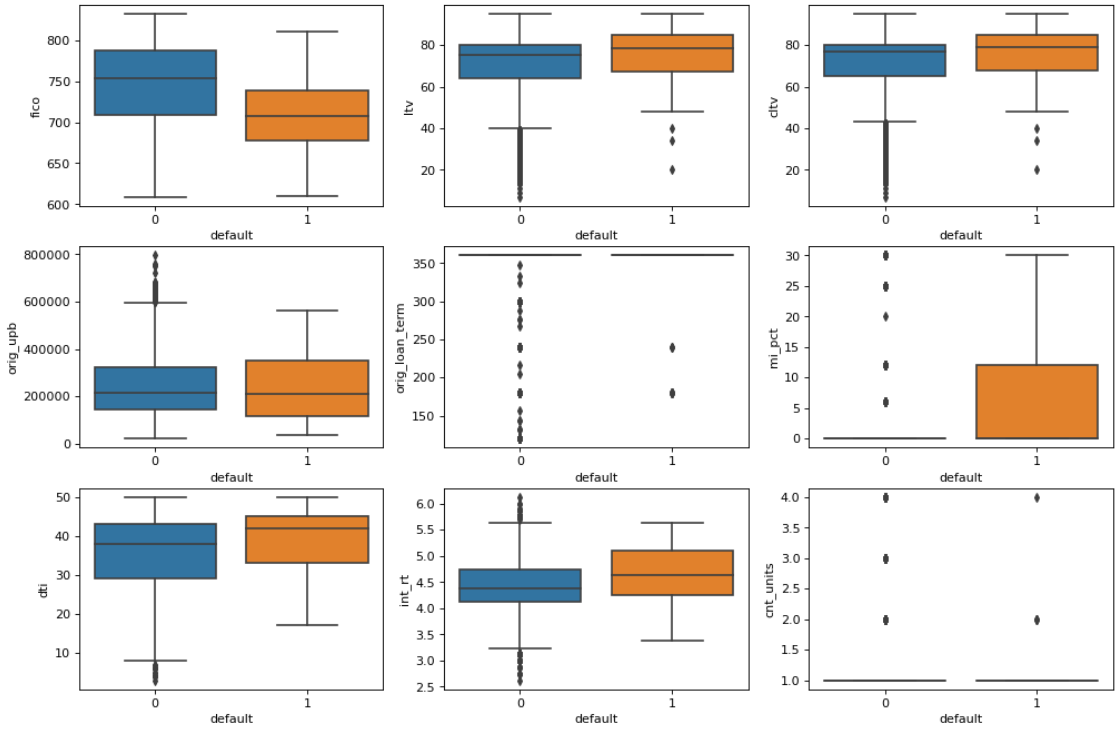
develop an understanding of which features could potentially serve as useful predictors for whether or not a mortgage was likely to be defaulted.

For the numerical features we created boxplots which displayed the differences in distribution of the features' values for default and non-default cases. For features in which there was noticeable differences in these two distributions, we concluded that the feature could be a useful indicator of whether a mortgage would be defaulted or not, and thus these features were kept in the clean dataset fo rthe model building process. On the other hand, in cases where there were no significant differences in the distributions of the feature between the default and non-default cases, we concluded that such features may not be useful in determining whether a mortgage was likely to default and thus we dropped them before beginning the construction of the model.

[11]:
```
#### DEFINE THE DIFFERENT TYPES OF VARIABLES ####
num_vars = ['fico','ltv','cltv','orig_upb','orig_loan_term','mi_pct',␣
 ↪'dti','int_rt', 'cnt_units', 'dt_fist_pi', 'dt']

#### STUDYING RELATIONSHIPS BETWEEN NUMERIC VARIABLES ####
fig, ax = plt.subplots(3, 3, figsize=(15,10))

for i, var in enumerate(list(num_vars[:9])):
    row, col = divmod(i,3)
    sns.boxplot(data=df_train_clean[df_train_clean[var] != 999], x='default',␣
 ↪y=var, ax=ax[row,col])
```



The above plots show that there is a slightly different distribution in the values of the features

'int_rt', 'ltv', 'cltv', 'dti', 'mi_pct' and especially 'fico' in the default cases compared to the non-default cases, thus kept these features for the model building phase as we felt they could be useful predictors for whether a loan would be defaulted or not. We also saw that there was not much difference in the distribution of the 'orig_upb' variable for default and non-default cases thus we decided to discard this feature.

[12]:
```
# Drop 'orig_upb' feature
df_train_clean = df_train_clean.drop(columns=['orig_upb'])
df_test_clean = df_test_clean.drop(columns=['orig_upb'])
```

On top of this however, we needed to investigate: * The difference between 'cltv' and 'ltv' to see if they encoded similar information and therefore determine whether we should discard one of these features, * The differences in the distributions of the 'orig_loan_term' and 'mi_pct' features for default and non-default cases using a different method than boxplots, as the plots shown above were quite hard to interpret.

We tackled the first issue by examining a correlation heatmap which showed the correlations between each of the features, and could thus be used to determine whether 'cltv' and 'ltv', as well as many other pairs of features, were highly correlated.

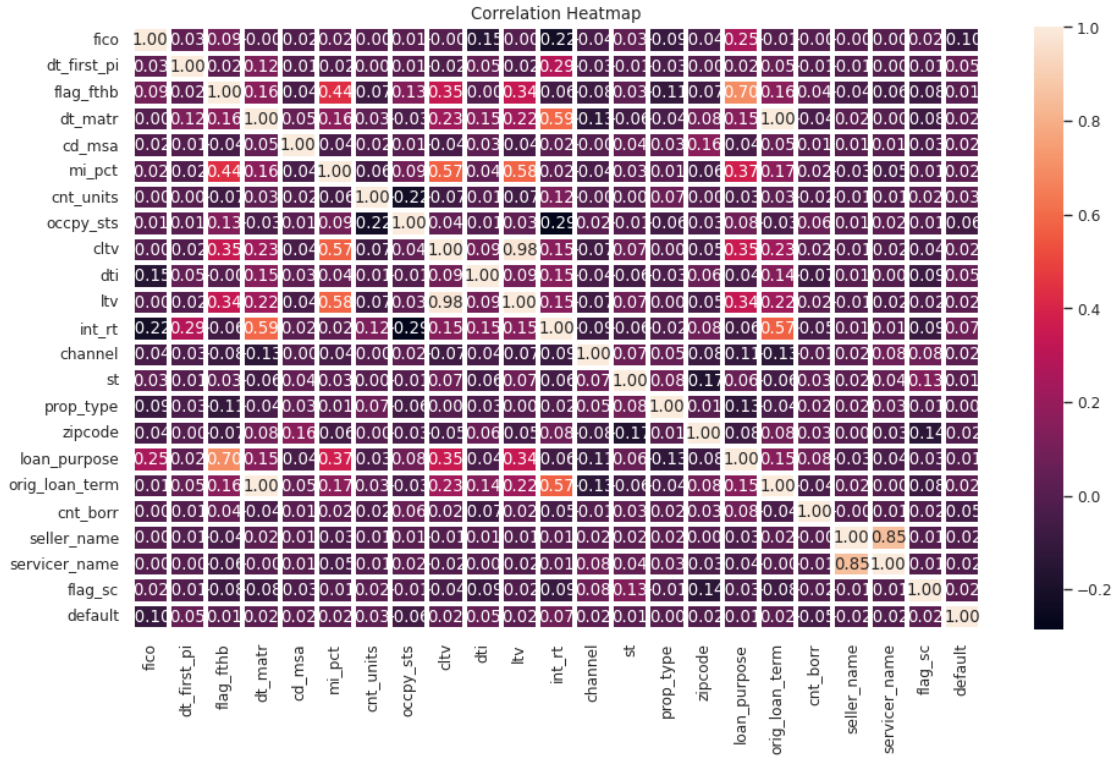
In order to produce this heatmap and as string data cannot be directly used for modelling in scikit-learn as it expects numerical inputs, we needed to encode the categorical variables into numerical form, thus we used a LabelEncoder to assign the categories a numerical value as this allows the same numerical value to be assigned to the same categories in both the training and test data. We also applied this LabelEncoder to the 'default' column in order to allow us to later call the labels for the pretty_confusion_matrix( ) function.

[13]:
```
# Store names for interpretations in discussion section.
df_train_clean_old_names = df_train_clean.copy()

# Assuming df_train_clean and df_test_clean are your DataFrames
# First, handle the 'default' column specifically
default_LE = LabelEncoder()
df_train_clean['default'] = default_LE.fit_transform(df_train_clean['default'])
df_test_clean['default'] = default_LE.transform(df_test_clean['default'])

# Then, loop through the rest of the object columns and encode them without␣
 ↪storing classes.
for column in df_train_clean.select_dtypes(include=['object']).columns:
    if column != 'default':  # Skip the 'default' column since it's already␣
 ↪encoded.
        LE = LabelEncoder()
        # Fit the encoder on the combined data from both DataFrames.
        LE.fit(pd.concat([df_train_clean[column], df_test_clean[column]],␣
 ↪axis=0, ignore_index=True))
        # Transform the columns in both DataFrames.
        df_train_clean[column] = LE.transform(df_train_clean[column])
        df_test_clean[column] = LE.transform(df_test_clean[column])
```

[14]:
```
# Plot heatmap of correlations.
sns.set(rc={'figure.figsize': (14, 8)})
sns.heatmap(df_train_clean.corr(), annot = True, fmt = '.2f', linewidths = 2)
plt.title("Correlation Heatmap")
plt.show()
```



The 'ltv' feature was described as the 'original loan-to-value' and 'cltv' was the 'original combined loan-to-value' meaning that they encode very similar information, and as expected these features had an extremely high correlation as can be seen in the correlation heatmap above. Due to this high correlation and the fact that the 'cltv' feature is more comprehensive since it includes all the loans on the property, not just the mortgage, we decided to drop the 'ltv' feature.

The correlation heatmap also shows that there is a 1:1 correlation between 'dt_matr' and 'orig_loan_term' which does make sense since they describe the 'maturity date of the loan' and the 'original loan term'. We decided to drop 'dt_matr' since 'orig_loan_term' is easier to work with as it describes the number of months the loan lasts rather than a date.

The correlation between the number of months of the loan 'orig_loan_term' and interest rate 'int_rt' is also fairly high at 0.57. In particular, after examining scatter plots of the data for these two features, we saw that loans with a longer duration have higher interest rates. However, since the information that is encoded by these features describes fairly distinct things, we still kept both of them, reducing the risk of overfitting the model to the nuances of our specific training data.

There is also a fairly high correlation of 0.57 between 'cltv' and 'mi_pct' which describes the 'mortgage insurance percentage'. In particular, Freddie Mac only insured mortgages that had a

'cltv' of around 80% or higher. Whilst this definitely suggests a high dependence of the value of 'mi_pct' on 'cltv', we again decided not to discard either of the two variables since they encoded very different pieces of information.

Finally, by reading the description of each of the features, we saw that the features 'cd_msa', 'zipcode' and 'st' all encoded information that pertained to the geographical location of the house to which the mortgage corresponded. Indeed in the project description, these features were described as follows:

- The 'cd_msa' feature describes 'METROPOLITAN STATISTICAL AREA (MSA) OR METROPOLITAN DIVISION: code, with null indicating that the area in which the mortgaged property is located is a) neither an MSA nor a Metropolitan Division, or b) unknown',
- The 'zipcode' feature describes 'The postal code for the location of the mortgaged property',
- The 'st' feature describes 'A two-letter abbreviation indicating the state or territory within which the property securing the mortgage is located.'

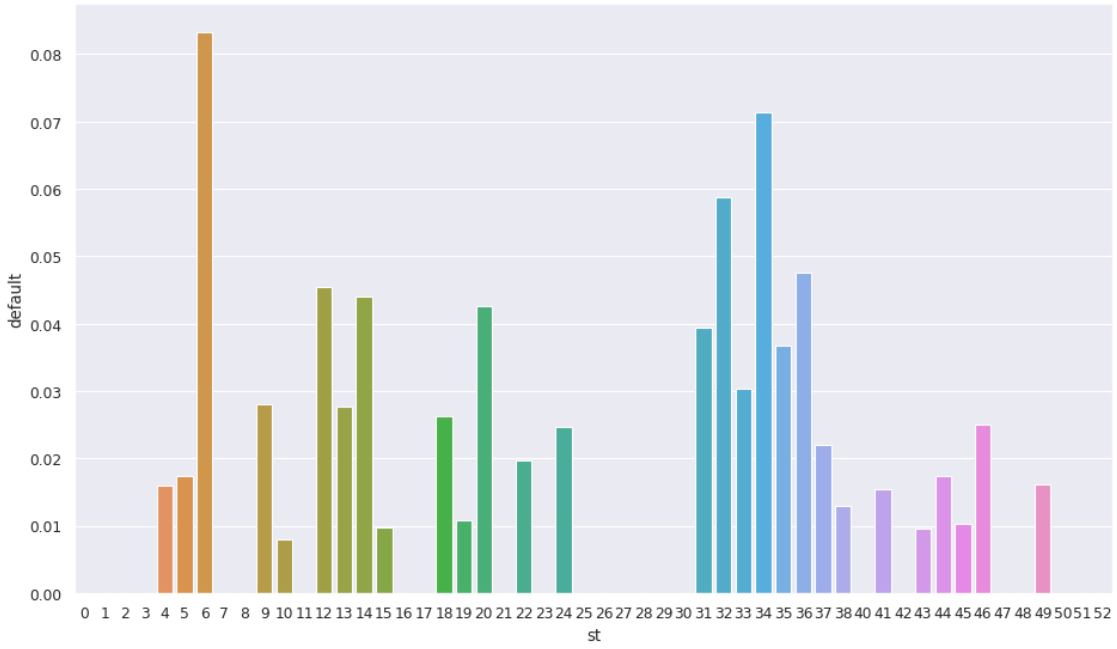
Due to the overlap in information which these features encode, we decided to only keep the 'st' feature since it has a significantly lower number of unique categories (only 52 compared to 343 and 662 for 'cd_msa' and 'zipcode' respectively) and thus significantly reduces the dimensionality of our data.

[15]:
```
df_train_clean = df_train_clean.drop(columns=['ltv','dt_matr'])
df_test_clean = df_test_clean.drop(columns=['ltv','dt_matr'])
# Count number of unique categories for 'cd_msa', 'zipcode', and 'st' features
df_train_clean.zipcode.nunique(), df_train_clean.cd_msa.nunique(),␣
 ↪df_train_clean.st.nunique()
df_train_clean = df_train_clean.drop(columns=['cd_msa','zipcode'])
df_test_clean = df_test_clean.drop(columns=['cd_msa','zipcode'])
```

Whilst the 'st' feature had significantly fewer categories than the 'zipcode' and 'cd_msa' features that we decided to remove from the dataset, it still had a separate category for each of the 52 states in the US, thus we decided to group together all the states in which there had never been an instance of a defaulted mortgage under the category of '99', thereby further reducing the number of categories. Additionally, to prevent the occurrence of unseen states in the test set, we merged less common (categories making up less than 3% of the dataset) categories into an overall category labelled '00'.

[16]:
```
# Create barplot showing default rates of each state
fig, ax = plt.subplots()
default_by_state = df_train_clean[['st','default']].
 ↪groupby(['st'],as_index=False).agg({'default':'mean'})
sns.barplot(default_by_state, x='st',y='default', ax = ax)
# Group together states with zero default rate under '99' to represent 'other'.
other = []
for i in default_by_state.st.unique():
    if (default_by_state[default_by_state.st == i]['default'].iloc[0] == 0):
        other.append(i)
df_train_clean['st'] = df_train_clean['st'].apply(lambda x: 99 if x in other␣
 ↪else x)
```

```
df_test_clean['st'] = df_test_clean['st'].apply(lambda x: 99 if x in other else␣
 ↪x)
```



[17]:
```
# Count occurrences of each unique value in 'st'
states_counts_train = df_train_clean['st'].value_counts()
states_counts_test = df_test_clean['st'].value_counts()
# Filter counts where occurrences are less than 3% of dataset.
threshold_train = 0.03*len(df_train_clean)
states_less_than_threshold_train = states_counts_train[states_counts_train <␣
 ↪threshold_train]
threshold_test = 0.03*len(df_test_clean)
states_less_than_threshold_test = states_counts_test[states_counts_test <␣
 ↪threshold_test]
# Get the number of 'st' values that are less than 3% of total dataset.
num_states_less_than_threshold_train = states_less_than_threshold_train.count()
print(f"Number of 'st' values that are less than 3% of total dataset:␣
 ↪{num_states_less_than_threshold_train}")
# Identify states to group into new other = 00 category.
states_to_group_train = states_counts_train[states_counts_train <␣
 ↪threshold_train].index
states_to_group_test = states_counts_test[states_counts_test < threshold_test].
 ↪index
# Replace 'st' column with grouped values in both training and test sets
df_train_clean['st'] = df_train_clean['st'].apply(lambda x: 00 if x in␣
 ↪states_to_group_train else x)
```

```
df_test_clean['st'] = df_test_clean['st'].apply(lambda x: 00 if x in␣
 ↪states_to_group_test else x)
```

```
Number of 'st' values that are less than 3% of total dataset: 21
```

Note that the 'seller_name' and 'servicer_name' also had categories for which there were no instances of defaults, but due to the much lower overall number of categories for both these features compared to the 'st' feature, we did not group them together.

To conclude, the features we used to create the prediction model, after data cleaning, were: * 'fico', 'dt_first_pi', 'flag_fthb', 'mi_pct', 'cnt_units', 'occpy_sts', 'cltv', 'dti', 'int_rt', 'channel', 'st', 'prop_type', 'loan_purpose', 'orig_loan_term', 'cnt_borr', 'seller_name', 'servicer_name', 'flag_sc', 'default'

4 Model Fitting and Tuning

To start the model creation, we began by separating the features from the target variable ('default'), and ensured the training and testing dataset had similar proportions of default cases. We then categorised the features into numerical and categorical types and identified their indices in the dataset, to be able to apply appropriate preprocessing transformations using the ColumnTransformer function provided by sklearn.

[18]:
```
# Split training data into target variable and features.
X_train = df_train_clean.drop("default", axis = 1)
y_train = df_train_clean["default"].copy()
X_test = df_test_clean.drop("default", axis = 1)
y_test = df_test_clean["default"].copy()

# Check that the same proportion of defaults exists in both the train and test␣
 ↪data sets.
print(pd.Series(y_train).value_counts(normalize=True)*100)
print(pd.Series(y_test).value_counts(normalize=True)*100)

# Separate feature names from values.
features = list(X_train.columns)

# Define Categorical and Numeric features and find their indexes in X_train and␣
 ↪X_test.
cat_feat =␣
 ↪['flag_fthb','cnt_units','occpy_sts','channel','st','prop_type','loan_purpose','cnt_borr',
            'seller_name','servicer_name','flag_sc']
cat_feat_cols = []
num_feat = ['fico','orig_loan_term']
num_feat_cols = []
cols_list = df_train_clean.drop(columns=['default']).columns.tolist()
for i in num_feat:
    num_feat_cols.append(cols_list.index(i))
for i in cat_feat:
```

```
    cat_feat_cols.append(cols_list.index(i))
```

```
0    98.159075
1     1.840925
Name: default, dtype: float64
0    98.138007
1     1.861993
Name: default, dtype: float64
```

[19]:
```
# Define the function from workshop 6 in order to print neat confusion matrices.
def pretty_confusion_matrix(confmat, labels, title, labeling=False,␣
 ↪highlight_indexes=[]):

    labels_list = [["TN", "FP"], ["FN", "TP"]]

    fig, ax = plt.subplots(figsize=(4, 4))
    ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
    for i in range(confmat.shape[0]):
        for j in range(confmat.shape[1]):
            if labeling:
                label = str(confmat[i, j])+" ("+labels_list[i][j]+")"
            else:
                label = confmat[i, j]
            if [i,j] in highlight_indexes:
                ax.text(x=j, y=i, s=label, va='center', ha='center',
                        weight = "bold", fontsize=18, color='#32618b')
            else:
                ax.text(x=j, y=i, s=label, va='center', ha='center')
    # change the labels
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        ax.set_xticklabels(['']+[labels[0], labels[1]])
        ax.set_yticklabels(['']+[labels[0], labels[1]])

    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    ax.xaxis.set_label_position('top')
    plt.suptitle(title)
    plt.tight_layout()

    plt.show()
```

Since the purpose of the model was to predict people who will default on mortgages, the choice of metric significantly depended on the cost of false negatives (FN) versus false positives (FP). Possible metrics we could have used for this model evaluation were the following:

1. False Positive Rate (FPR):

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

- Measures the proportion of non-defaulters that are incorrectly classified as defaulters.
- A high FPR means that many customers who would not default are predicted to do so, which can lead to unnecessary actions that could negatively impact customer relationships or lead to loss of business.

2. True Positive Rate (Recall):

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

- Measures the proportion of actual defaulters correctly identified by the model.
- High recall is crucial in this scenario because failing to identify a defaulter (FN) can have significant financial implications.

3. Precision:

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

- Measures the proportion of predicted defaulters who actually default.
- While still important, in this context precision alone may not be the best metric because a model can achieve high precision by being overly cautious in predicting defaulters (i.e., predicting very few people as defaulters, but most of those predictions are correct). This could lead to unecessarily large numbers of defaulters being missed, which could have significant financial implications.

4. Accuracy:

$$\text{Accuracy } = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

- Measures the proportion of all predictions (both defaulters and non-defaulters) that are correctly identified by the model.
- Not ideal in the context of a highly imbalanced dataset, as then accuracy can be deceptive. A model could predict 'no default' for all cases, ignoring the less frequent but critical 'default' cases, and still appear highly accurate due to the large number of true negatives (TN).

In the case of mortgage defaults the negative consequences of incorrectly classifying an individual who will default far outweigh those of incorrectly classifying an individual who will not default, thus whilst the latter should still ideally be avoided, our priority was to the number of false negatives.

As a result, a high accuracy rate was not necessarily indicative that our model was performing effectively, so instead we decided to focus on the 'Recall' metric when ranking and selecting models. However, it must be noted that a perfect recall (a score of 1) can result from classifying every observation as a default, a strategy that lacks practicality. So, to maintain a balanced perspective

without disregarding precision, we introduced a 'custom_scorer'. This scoring approach significantly weighted recall while still factoring in precision, and ensured that while our focus remains on correctly identifying defaults, we also keep false alarms to a manageable level.

[20]:
```
def custom_score(y_true, y_pred):
    """
    Custom scoring function that prioritises recall but also takes precision␣
 ↪into account.
    """
    recall_weight = 0.7 # More weight on recall because false negatives are␣
 ↪more costly.
    precision_weight = 0.3 # Some weight on precision to balance the␣
 ↪performance.

    # Calculate custom score with more emphasis on recall.
    recall = recall_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred)
    score = (recall_weight * recall) + (precision_weight * precision)
    return score

# Make the scorer object.
custom_scorer = make_scorer(custom_score, greater_is_better=True)
# Define a function to calculate the recall.
def recall_fun(confmat):
    return confmat[1,1]/(confmat[1,0] + confmat[1,1])
```

To allow us to compare the various prediciton models we produced and tested, we created a dataframe 'df_Model_Performance' in which we stored all model performance metrics for later discussion.

[51]:
```
# Create the empty DataFrame.
df_Performance_Metrics = pd.DataFrame(columns=["Model", "Oversampling", "C",␣
 ↪"MSE", "AUC", "Recall"])
```

In testing a simple logistic regression model without oversampling for regularisation, we found that it could not cope with the massively imbalanced dataset, and as a result predicted that no cases at all would default. Clearly refinement was needed to deal with the imbalanced data issue. This issue could be dealt with in many ways, some of which are described below.

1. Changing the regularization strength (C).

2. Weighting the classes in the model during training:

During model fitting we can assign a larger penalty to wrong predictions on the minority class either manually or using Scikit-Learn.

- In Scikit-Learn:

  Scikit-Learn offers a built-in option 'class_weight="balanced"' to automate this weighting based on the class distribution in the data. The formula used to calculate the weight of each class is:

$$\frac{n}{N_c \times \sum_{i=1}^{n} I(y_i \in S)}$$

where n are the number of samples, N_c the number of classes, I is an indicator function, and S contains the class

- Manually:

  There is also the option to incorporate manual class weight adjustments into the model training. We can define a range of weight options allowing for the experimentation of different weighting schemes to find the one that optimally balances precision and recall, particularly for the minority class. To do this we can create a list of dictionaries, with each dictionary specifying different weights for the classes. For example we later use 'class_weight_options = [{0: 1, 1: v} for v in np.linspace(1.1, 1.5, 5)]'.

3. Resampling the data:

We can resample to change the distribution of the classes in our training data. This can be done in two ways.

- Over-sampling:

  Over-sampling increases the size of the minority class(es) to balance the class distribution. The simplest form of over-sampling is to randomly replicate instances of the minority class in the dataset until the class distribution is more balanced. However, this can lead to overfitting, as the model might learn to recognize specific instances rather than general patterns.

- Under-sampling:

  Under-sampling reduces the size of the majority class(es) to balance the class distribution. The simplest form of under-sampling involves randomly removing instances from the majority class until the class distribution is more balanced. However, this can lead to a loss of information, which might decrease the model's performance.

We tried and tested all of these techniques before settling on the following gridsearched logistic regression model, which we later applied a threshold to.

[21]:
```
# Define a Logistic Regression pipeline including oversampling.
Logistic_Pipeline = ImPipeline([
    ("pre_processing", ColumnTransformer([
        ("num_pre", StandardScaler(), num_feat_cols),
        ("cat_pre", Pipeline([
            ("one_hot", OneHotEncoder(drop='first', handle_unknown='ignore')),
            ("scaler", StandardScaler(with_mean=False))
        ]), cat_feat_cols)
    ])),
    ('oversample', RandomOverSampler(random_state=seed)),
    ("model", LogisticRegression(random_state=seed))
])

# Define the parameter grid.
parameter_grid = {
```

```
    'model__C': [2**i for i in range(-5, 10, 2)],
    'model__class_weight': [{0: 1, 1: v} for v in np.linspace(1.1, 2, 10)],
}

# Now, use GridSearchCV to select the best model.
Log_CV = GridSearchCV(
    Logistic_Pipeline,
    param_grid=parameter_grid,
    scoring=custom_scorer,
    refit=True,
    cv=StratifiedKFold(10, shuffle=True, random_state=seed),
    n_jobs=-1, # Use all processors.
    return_train_score=True
)

# Fit the model.
Log_CV.fit(X_train, y_train)
```

[21]:
```
GridSearchCV(cv=StratifiedKFold(n_splits=10, random_state=None, shuffle=True),
             estimator=Pipeline(steps=[('pre_processing',
ColumnTransformer(transformers=[('num_pre',
StandardScaler(),
                                                                [0,
                                                                 13]),
('cat_pre',
Pipeline(steps=[('one_hot',
          OneHotEncoder(drop='first',
                        handle_unknown='ignore')),
         ('scaler',
          StandardScaler(with_mean=False))]),
                                                                [2, 4,
                                                                 5, 9,
                                                                 10,
                                                                 11,
                                                                 12,
                                                                 14,
                                                                 15,
                                                                 16…
                                      ('oversample', RandomOverSampler()),
                                      ('model', LogisticRegression())]),
             n_jobs=-1,
             param_grid={'model__C': [0.03125, 0.125, 0.5, 2, 8, 32, 128, 512],
                         'model__class_weight': [{0: 1, 1: 1.1},
                                                 {0: 1, 1: 1.2000000000000002},
                                                 {0: 1, 1: 1.3},
                                                 {0: 1, 1: 1.4000000000000001},
                                                 {0: 1, 1: 1.5}, {0: 1, 1: 1.6},
```

```
                                                {0: 1, 1: 1.7000000000000002},
                                                {0: 1, 1: 1.8}, {0: 1, 1: 1.9},
                                                {0: 1, 1: 2.0}]},
                   return_train_score=True, scoring=make_scorer(custom_score))
```

As was mentioned earlier, we chose to further enhance the decision-making process for our model by setting a threshold on the predicted probabilities ('y_probs_...' vectors below) for classifying each test instance (X_test) as either default or non-default. Specifically, 'y_probs_...' denotes the likelihood of each instance in the test dataset being a default case. By applying a specific threshold to these probabilities, we determined the classification outcome for each instance: any instance with a predicted probability equal to or exceeding this threshold was marked as '1' (indicating default), whereas instances with probabilities below the threshold were marked as '0' (indicating non-default).

To optimally set this threshold, we analysed the trade-off between False Positive (FP) and False Negative (FN) rates across various threshold levels in a plot. Through this analysis we aimed to strike a balance between accurately identifying actual defaults (minimising FNs) and avoiding an overestimation of defaults within the non-defaulting population (minimising FPs).

[74]:
```
# Predict probabilities.
y_probs_log = Log_CV.predict_proba(X_test)[:, 1]

# We base the plot off of the logistic model.
# Generate a range of thresholds from min to max probability.
thresholds = list(np.linspace(start=min(y_probs_log), stop=max(y_probs_log),␣
 ↪num=100))

# Initialise lists to store false positive and false negative rates.
fp_rates = []
fn_rates = []

for thresh in thresholds:
    # Apply threshold to predictions to get binary outcome.
    y_pred_thresh = (y_probs_log >= thresh).astype(int)

    # Calculate confusion matrix and get important values.
    conf_matrix = confusion_matrix(y_test, y_pred_thresh)
    TN, FP, FN, TP = conf_matrix.ravel()

    # Calculate rates.
    fp_rate = FP / (FP + TN)
    fn_rate = FN / (FN + TP)

    fp_rates.append(fp_rate)
    fn_rates.append(fn_rate)

# Plot the rates.
plt.figure(figsize=(10, 5))
```

```
plt.plot(thresholds, fp_rates, label='False Positive Rate')
plt.plot(thresholds, fn_rates, label='False Negative Rate')

# Using this plot we chose the following threshold.
Our_Threshold = 0.75

# Add vertical dashed line at Our_Threshold.
plt.axvline(x=Our_Threshold, color='r', linestyle='--', label='Our Threshold')

plt.xlabel('Thresholds')
plt.ylabel('Rate')
plt.title('Threshold vs False Positive/Negative Rates')
plt.legend()
plt.show()

# Adjust the prediction based on the new threshold.
y_pred_adjusted_log = (y_probs_log >= Our_Threshold).astype(int)
```

This plot shows that as the threshold for predicting the positive class increases, the False Positive Rate (FPR) decreases while the False Negative Rate (FNR) increases.

Lower Thresholds: - Lead to a higher FPR, meaning the model is more likely to incorrectly classify negative instances as positive. This is because a lower threshold makes it easier for an instance to be predicted as the positive class. - Result in a lower FNR, meaning the model is less likely to miss the positive instances. This is due to the "easier" criteria for predicting positives; thus, fewer actual positives are overlooked.

Higher Thresholds: - Reduce the FPR, meaning that the model is more conservative with positive

class predictions. It is less likely to label a negative instance as positive because the criteria are stricter. - Increase the FNR, indicating that the model is more likely to miss out on true positive instances. With a higher bar for classifying positives, some true positives don't meet the criteria and are therefore classified as negatives.

At any chosen threshold there was a certain trade-off between being too lenient and too strict with the classifier's threshold. We wanted to select a threshold that balanced the consequences of false positives against false negatives. The optimal threshold was the one that minimises the overall cost or risk associated with these two types of errors, which varied depending on the specific costs of a mortgage default and the business's risk appetite. Therefore the business may choose to vary the specific threshold to match their own operating criteria. In the end we chose a threshold of 0.75 because it struck an appropriate balance, aligning with our primary objective, within the context of mortgage defaults, of minimising the number of false negatives.

5 Discussion & Conclusions

We evaluated several models, detailed below, with key parameters optimized through GridSearchCV. All models utilised standard scaling for all features.

- Baseline Model: Implemented as a straightforward logistic regression with no regularization (penalty=None), defaulting to scikit-learn's standard C value of 1.0. No parameter tuning was conducted.
-
- Model 1 (Oversampling Logistic): Optimized for the C parameter only.
- Model 2 (Balanced Oversampling Logistic): Similar to Model 1 with the addition of the "class_weight="balanced" ' setting, chosen from options [None, "balanced"], to account for class imbalance.
- Model 3 (GridSearched SVC): Extensive parameter tuning included kernel type, kernel parameters, C value, and class weight. This model was computationally intensive, taking over 5 minutes to execute, and ultimately chose an rbf kernel with gamma set to 0.001 and a class weight ratio of 1:1.23.
- Model 4 (Gridsearched Logistic): Tuned for both C and class weight, with an optimal class weight determined to be 1.9.
- Models 5 and 6: Adjustments involved applying a probability threshold to the predictions from Models 3 and 4, respectively, retaining their parameter settings.

The performance metrics for each tested model alongside whether or not oversampling was used and each model's respective chosen C value are all detailed are detialed in the table below.

[76]:
```
df_Performance_Metrics
```

[76]:

	Model	Oversampling
0	Baseline Logistic	No
1	Oversampling Logistic (with optimal C)	Yes
2	Oversampling Balanced Logistic (with optimal C)	Yes
3	GridSearched SVC	Yes

4	GridSearched Logistic	Yes
5	SVC (with threshold)	Yes
6	GridSearched Logistic (with threshold)	Yes

	C	MSE	AUC	Recall
0	1.0 (Default)	0.020263	0.743025	0.0
1	0.106	0.277656	0.739184	0.588235
2	0.236	0.271632	0.737871	0.617647
3	0.313	0.557503	0.743189	0.852941
4	0.313	0.342278	0.793494	0.852941
5	0.313	0.114458	0.7431886817226891	0.411765
6	0.313	0.152245	0.7934939600840336	0.558824

Choosing the best model for predicting mortgage defaults from the options detailed in the table involves considering several factors based on the metrics calculated: Mean Squared Error (MSE), Area Under the Curve (AUC) and Recall.

In the context of mortgage default prediction:

- A Low MSE is desirable, indicating that the model's predictions are close to the actual outcomes.
- A High AUC is critical as it suggests the model's ability to distinguish between those who will default and those who will not.
- High Recall is important if the cost of false negatives (i.e., failing to identify a default) is substantial (which is typically the case in financial settings).

Considering the calculated metrics:

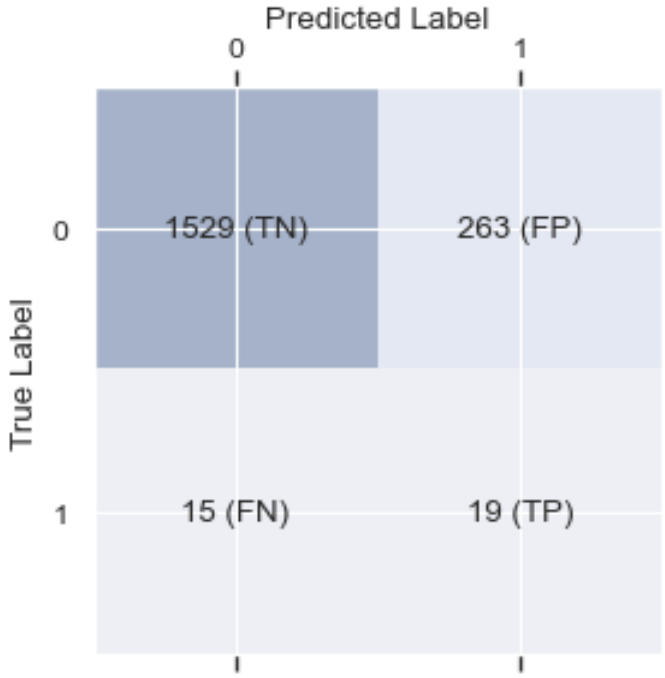
- Models 1 and 2 (Oversampling Logistic with optimal C) have higher MSE compared to the baseline but have significantly better recall, which could be crucial for a default prediction context where missing a defaulting loan is very costly.
- Models 4 and 6 (GridSearched Logistic and GridSearched Logistic (with threshold)) show an improved AUC compared to the baseline, indicating better discriminative ability, and model 6 also shows a very strong balance with a much-reduced MSE after applying a threshold.
- Model 5 (SVC with threshold) shows the lowest MSE and a reasonable AUC, although the recall is significantly lower, indicating it might miss many actual defaults.

For Freddie Mac, Model 6 is recommended based on several factors:

- It's a logistic regression model, which ensures computational efficiency compared to more complex models like SVC, and ease of interpretability, with its coefficients being easily accessible.
- The employment of GridSearchCV in Model 6 allows for automatic tuning to optimal parameters, which is beneficial for adaptability to possible new training data, in the future.
- Model 6 strikes a balance between predictive accuracy (as shown by MSE) and classification capability (as evidenced by AUC), which is indicative of strong overall performance.
- Crucially, the incorporation of a custom scoring function and an adjustable threshold means that the model can be easily adapted and fine-tuned to align with the business's changing objectives, financial conditions, and risk tolerance.

The confusion matrix for this model is shown below.

[77]:
```
pretty_confusion_matrix(Confusion_Mat_6, default_LE.classes_, "",  labeling =␣
↪True)
```



The confusion matrix shows multiple desirable properties including:

- High True Negative Rate. The model is quite effective at identifying the negative class, with a large number of true negatives (1529).
- Low False Negative Rate. There are very few false negatives (15), suggesting that the model rarely misses the positive cases. Which in the setting of mortgage default detection, having a low false negative rate is crucial.
- Moderate False Positive Rate. There is a moderate number of false positives (263), indicating that the model sometimes incorrectly predicts the positive class, however in the context of mortgage defaults where the data is incredibly imbalanced and having a low false negative rate is the priority, this may be a price worth paying.
- Good Ratio of True Positives to False Negatives. Again this identifies that the model is good at identifying the defaulting cases.

As mentioned above, Model 6 is easily intepretable with easily accesible coefficients. We extract and intepret these below allowing us to make evidence backed recomendations to the Freddie Mac.

As previously noted, Model 6 offers the advantage of interpretability through easily accessible coefficients. Below, we extract these coefficients to provide data-driven insights and recommendations for Freddie Mac.

In a logistic regression model that implements standardisation, each coefficient represents the change in the log odds of the outcome per one standard deviation increase in the corresponding original variable, holding all other features constant. Thus we can convert these standardised log odds to standardised odds, by exponentiating each coefficient. These odds now indicate the factor by which the likelihood of the outcome is multiplied for each one standard deviation increase. Thus odds below 1 signify a reduction in likelihood with each standard deviation increase in the feature, while odds above 1 signify an decrease.

To understand the model's predictions, we calculated these odds ratios. We then categorised the features into those that decrease the predicted likelihood of the outcome (odds < 1) and those that increase it (odds ≥ 1). By sorting these categories by the absolute values of the coefficients, we can rank the features by their influence on the outcome.

In terms of directly interpreting the odds, for example, odds of 0.8 indicate that the predicted probability of the outcome decreases by 20% for each one standard deviation increase in the corresponding original variable, assuming all other variables are held constant.

[22]:
```
# Access the fitted pipeline from the best_estimator_ of the GridSearchCV␣
↪object.
fitted_pipeline = Log_CV.best_estimator_
# Access the pre-processing step.
pre_processing = fitted_pipeline.named_steps['pre_processing']
# Access the OneHotEncoder directly from the named_transformers_ of the␣
↪ColumnTransformer.
one_hot_encoder = pre_processing.named_transformers_['cat_pre']
# Get feature names for the one-hot encoded categorical features.
encoded_feature_names = one_hot_encoder.get_feature_names_out()
# Map feature indices to column names.
num_feature_names = num_feat
cat_feature_names = encoded_feature_names  # These are now correctly obtained␣
↪from the one-hot encoder
# Combine numeric and categorical feature names.
all_feature_names = num_feature_names + list(cat_feature_names)
# Access the model.
Logistic_Model = fitted_pipeline.named_steps['model']
# Access the best estimator from GridSearchCV.
best_pipeline = Log_CV.best_estimator_
# Access the logistic regression model from the pipeline.
logistic_model = best_pipeline.named_steps['model']
# Get the coefficients from the logistic regression model.
coefficients = Logistic_Model.coef_
# Flatten the coefficients array to match the feature names' dimensionality
coefficients = coefficients.flatten()
odds = np.exp(coefficients)
# Create a DataFrame with feature names and their corresponding coefficients
coef_df = pd.DataFrame({
    'Feature': all_feature_names,
    'Coefficient': coefficients,
```

```
    'Standardised Odds': odds
})
# Split the DataFrame based on the odds being less than 1 or greater than/equal␣
 ↪to 1
# DataFrame where the odds are less than 1.
less_than_one_df = coef_df[coef_df['Standardised Odds'] < 1].copy()
# DataFrame where the odds are greater than or equal to 1.
greater_than_or_equal_one_df = coef_df[coef_df['Standardised Odds'] >= 1].copy()
```

[23]:
```
# Sort the DataFrame by the absolute value of coefficients, from largest to␣
 ↪smallest, taking top 10.
greater_than_or_equal_one_df = greater_than_or_equal_one_df.
 ↪reindex(greater_than_or_equal_one_df['Coefficient'].abs().
 ↪sort_values(ascending=False).index).head(10)
# Reset the index to get a clean DataFrame.
greater_than_or_equal_one_df = greater_than_or_equal_one_df.
 ↪reset_index(drop=True)
greater_than_or_equal_one_df
```

[23]:

	Feature	Coefficient	Standardised Odds
0	seller_name_18	0.636213	1.889313
1	flag_fthb_2	0.622632	1.863827
2	flag_fthb_1	0.612523	1.845081
3	servicer_name_19	0.374682	1.454528
4	servicer_name_30	0.345339	1.412469
5	loan_purpose_1	0.333842	1.396323
6	prop_type_4	0.291546	1.338495
7	flag_sc_1	0.217947	1.243521
8	st_14	0.194790	1.215056
9	orig_loan_term	0.180156	1.197404

[24]:
```
# Do same as above for other dataframe.
less_than_one_df = less_than_one_df.reindex(less_than_one_df['Coefficient'].
 ↪abs().sort_values(ascending=False).index).head(10)
less_than_one_df  = less_than_one_df.reset_index(drop=True)
less_than_one_df
```

[24]:

	Feature	Coefficient	Standardised Odds
0	st_99	-1.619754	0.197947
1	fico	-0.992642	0.370596
2	occpy_sts_1	-0.887769	0.411573
3	seller_name_27	-0.863038	0.421878
4	servicer_name_13	-0.644356	0.525000
5	servicer_name_18	-0.607355	0.544790
6	seller_name_25	-0.583989	0.557670
7	occpy_sts_2	-0.548097	0.578049
8	servicer_name_25	-0.492130	0.611323

```
9          cnt_borr_2    -0.449771            0.637774
```

With the exception of 'fico' the predominant influential features from both dataframes were all categorical. Within our processing pipeline, these features underwent one-hot encoding prior to being standardised using the StandardScaler function, specifically set with 'with_mean=False' to accommodate sparse data from the encoding. This simple preprocessing allows for a straightforward translation of the coefficients back to the original data scale. To achieve this, we simply multiplied each coefficient by its corresponding standard deviation.

Since one-hot encoded features are binary (0 or 1), their standard deviation can be calculated from the training data after encoding. Since these variables take on values of 0 or 1, their variance (and thus standard deviation) reflects the proportion of $1s$ in the data. The formula for the standard deviation of a binary variable is:

$$\sigma = \sqrt{p(1-p)}$$

where p is the proportion of data points that are 1 for a given feature.

Thus, in order to interpret the coefficients of these influential features in terms of the original data, we can calculate this quantity for any given feature and multiply it by its corresponding coefficient. We include an example for the first row of the 'less_than_one_df' dataframe.

[25]:
```
n = len(df_train_clean)
# Count the occurrences of value in the corresponding d_train column.
count = (df_train_clean['st'] == 99).sum()
# Calculate proportion.
p = count / n
# Calculate standard deviation.
sigma = (p * (1 - p)) ** 0.5
# Get the unstandardised coeffcient
true_coefficient = less_than_one_df['Coefficient'].iloc[0]*sigma
# Get the unstandardised odds.
true_odds = np.exp(true_coefficient)
print(f'After reversing the standardisation process, we observed that in our␣
 ↪model the probability of default if an individual resides in the 99 state␣
 ↪category (a classification for states without any recorded defaults)␣
 ↪decreased by {round((1-true_odds)*100, 2)}%, holding all other variables␣
 ↪constant.')
```

```
After reversing the standardisation process, we observed that in our model the
probability of default if an individual resides in the 99 state category (a
classification for states without any recorded defaults) decreased by 46.57%,
holding all other variables constant.
```

Alternatively for numerical features such as 'fico'that have been standardised using the 'Standard-Scaler' function, the interpretation of the coefficients in terms of the original, non-standardised data is given using the following translation:

$$\text{Change in odds per unit increase in X } = \exp(\beta \times \sigma)$$

Where:

- β is the standardised coefficient for a standardised feature.
- X is an original variable in the dataset.
- σ is the standard deviation of the original variable.

We note that this does not incorporate the mean of the feature, and it does not need to for the interpretation of the coefficients themselves. Its simply the standard deviation that is used to rescale the coefficient to show the change in log odds for a one unit increase in the original variable.

We further note that for each categorical variable featured in the above dataframe we can easily identify which category each number corresponds to by reading off the value appearing at that index in the list of unique values from that column in our original dataset. An example of this is given below.

[87]:
```
df_train_clean_old_names['seller_name'].unique()[18]
```

[87]:
```
'FIFTHTHIRDMTGECO'
```

After categorising factors into those which correlate the strongest with defaulting and not defaulting, we are able to analyse these factors in order to make actionable decisions with regards to loan lending.

The factors which exhibited the strongest positive correlation with defaults were:

- "seller_name" representing the vendor selling the loan to Freddie Mac,
- "flag_fthb" representing whether the receipent of the loan is a first time house buyer,
- "servicer_name" representing the servicer of the mortgage loan to Freddie Mac,
- "prop_type" representing the specific type of property,
- "loan_purpose" representing the reason a borrower is seeking their mortgage loan, specifically a Cash-out Refinance mortgage (C), and
- "st" representing the state or territory in which the mortgage is located, specifically the state of Oregon.

The sellers representing the strongest positive correlation with defaults were sellers 18, 5, 13 and 1, representing "FIFTHTHIRDMTGECO", "GUARANTEEDRATE,INC", "PRIMELENDING,APLAINS" and "WELLSFARGOBANK,NA" respectively.

The recipients of those loans under "flat_fthb" which correlate strongest with defaults are those first-time house buyers that intened to reside in the mortgaged property as a primary residence.

The servicers which showed the strongest positive correlation with defaults were Servicers 12, 3 and 19 representing "AURORAFINANCIALGROUP", "MATRIXFINANCIALSERVI" and "PINGORALOANSERVICING" respectively.

The property type showing strong positive correlations with defaults were proposed unit development properties.

The factors exhibiting the strongest negative correlation with defaults were:

- "st", specifically the state "99", which is the collection of states across the dataset which did not contain a default,
- "fico", a number prepared by the third-party, FICO, representing the credit score of the borrower,
- "occpy_sts" representing the occupier state of the property, specifically, "owner occupied" corresponding to value "1" and "second-home" corresponding to value "2",
- "seller_name", specifically, seller "27": "CITIZENSBANK, NATL", Seller "25": "NATION-STARMTGCLLC" and Seller "14": "FLAGSTARBANK, FSB",
- "servicer_name" representing specifically Servicer "13": "USBANKNA" and Servicer "18": "SUNTRUSTBANK", and
- "cnt_borr" representing the number of borrowers associated with the mortage loan securedby the property, specifically "2": more than one borrower.

We see, for example, that the most favourable attributes we would like to see in subjects are they have a 'yes' in the feature 'st_99' which indicates they are from one of the states that have no prior defaulting history, this of course aligns with expectations.

Overall we would especially like to see more future work with a number of specific sellers and servicers, as well as subjects from states with no defaulting history. We also prefer to see 'occpy_sts_1' and 'occpy_sts_2' indicating more experienced buyers or investors. As may have been expected, one sees lower defaulting rates amongst subjects with higher 'fico' scores, which is already a well understood measure of how likely a subject is to meet financial agreements and repayments. We also like to see a 'cnt_borr' value of 2, indicating that having more than 1 subject responsible for paying a loan decreases the likelihood of defaulting on the payment.

Conversely, from the chart with odds greater than 1, we also have a list of sellers and servicers names that we may encourage to take more precuations with before working with those venders. Additionally, we see that we may want to be cautious with subjects taking a value of 'Y' in the feature 'flag_fthb', which indicates that a subject has been flagged as a potential first time homebuyer, suggesting that first time buyers are more likely to default based on our model.

6 References

[1] Freddie Mac. (2023). Single Family Loan-Level Dataset. Freddie Mac. https://www.freddiemac.com/research/datasets/sf-loanlevel-dataset

[1]:
```
# Run the following to render to PDF
!jupyter nbconvert --to pdf Project2.ipynb
```

```
[NbConvertApp] Converting notebook Project2.ipynb to pdf
[NbConvertApp] Writing 460872 bytes to Project2.pdf
```

[ ]:
```

```