project2prev

April 15, 2025

1 Machine Learning in Python - Project 2

Alfie Plant, Markus Emmott, Oscar Youngman, Ashe Raymond-Barker

[4]:
```
# Data libraries
import pandas as pd
import numpy as np
from math import isnan
# Plotting libraries
import matplotlib.pyplot as plt
from IPython.display import Image
import seaborn as sns
# Plotting defaults
plt.rcParams['figure.figsize'] = (8,5)
plt.rcParams['figure.dpi'] = 80
import keras
import tensorflow as tf
# sklearn modules
import sklearn
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC, LinearSVC
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder,␣
 ↪KBinsDiscretizer
from sklearn.preprocessing import Binarizer, PolynomialFeatures, MinMaxScaler
from sklearn.pipeline import make_pipeline
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV, KFold, StratifiedKFold
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix
from sklearn.metrics import accuracy_score, precision_score, f1_score,␣
 ↪recall_score
from sklearn.metrics import RocCurveDisplay, roc_auc_score
from sklearn.metrics import PrecisionRecallDisplay
# EDA plotter functions
from helper_funcs import *
# from geo_plotting_functions import *
```

```
from imblearn.over_sampling import RandomOverSampler, SMOTE
from imblearn.under_sampling import RandomUnderSampler
```

2 Introduction

This report will outline the approaches made to produce a model that accurately distinguishes between Freddie Mac Fixed-Rate Mortgage loans that have defaulted and those that do not. The model will be used to offer insight on active loans that may be at risk of default in the future, but also to identify different factors that have an impact on defaulting. Our model produces synthetic data so that there is balance in the number of defaults versus prepaid loans, ensuring that there is similar weight placed on correctly identifying a default compared with a prepaid loan. We have explored various linear and non-linear models. Linear models serve as an intuitive way to interpet results, but due to their simplicity, do not capture the non-linear relationships present in the data. Random Forest and Neural Networks have been explored in more detail and these provide stronger results. Our final model is a Neural Network model, which outperforms the Random Forest, when we prioritise balancing identification. However, we have included a detailed breakdown of feature importance using the Random Forest, which is easily interpretable for the company to determine which factors contribute the most to defaulting mortgages. We have also used our final model to examine active Freddie Mac loans to provide extra insight on loans that are at risk of defaulting.

2.0.1 Data Overview

The dataset consists of 200,000 fully amortizing fixed-rate Single-Family mortgages. Of these, 125,959 have been paid off, 73,295 are still active, and 746 have defaulted. There are 32 features, which can be categorised as follows:

- **Personal Details:**
  - **`cnt_borr`** (Categorical): Whether one of multiple borrowers are obligated to pay the mortgage: One borrower (1), Multiple borrowers (2)
  - **`fico`** (Numeric): Borrower's credit score
  - **`dti`** (Numeric): Ratio value of borrower(s) monthly debt payments to total monthly income
  - **`cltv`** (Numeric): Ratio value of all combined loans of Borrower(s) to value of property
  - **`ltv`** (Numeric): Ratio value of mortgage loan of Borrower(s) to value of property
- **Loan Details:**
  - **`channel`** (Categorical): Indicates whether Broker or correspondent was involved in origination of mortgage loan: Retail(R), Broker(B), Correspondent(C)
  - **`loan_purpose`** (Categorical): Indicates type of loan: Chash-out refinance mortgage(C), No cash-out refinance mortgage(N), Purchase mortgage(P)
  - **`prod_type`** (Categorical): Denotes whether mortgage is fixed rate or adjustable rate
  - **`seller_name`** (Categorical): Entity acting as seller of mortgages to Freddie Mac
  - **`servicer_name`** (Categorical): Entity acting as servicer of mortgages to Freddie Mac
  - **`flag_sc`** (Categorical): Indicates whether mortgages with origination dates after 10/1/2008, delived to Freddie Mac on or after 1/1/2009 exceed the conforming loan limits
  - **`program_ind`** (Categorical): Indicates whether loan is part of any programmes: Home Possible(H), HFA Advantage(F), Refi Possible(R) Not Available/Applicable(9)
  - **`rr_ind`** (Categorical): Indicates whether loan is part of Relief Refinance Program.

    - **io_ind** (Categorical): Indicates whether loan only requires interest payments for a specific period
    - **mi_cancel_ind** (Categorical): Indicates whether mortgage insurance has been reported as cancelled after time of Freddie Macs purchase of the mortgage loan: Cancelled(Y), Not Cancelled(N), Not Applicable(7), Not Disclosed(9)
    - **loan_status** (Categorical): Indicates whether borrower(s) has defaulted on loan, paid off loan or if loan is still active
    - **dt_first_pi** (Numeric): First payment date of mortgage (YYYYMM)
    - **dt_matr** (Numeric): Final payment date of mortgage (YYYYMM)
    - **mi_pct** (Numeric): Percentage of loss coverage on the loan insurance plan, where 0 indicates no insurance
    - **orig_upb** (Numeric): Unpaid balance of mortgage to the nearest $1,000
    - **int_rt** (Numeric): Interest rate of loan
    - **orig_loan_term** (Numeric): Number of scheduled monthly repayments on mortgage
    - **id_loan** (Numeric): Unique loan identifier that includes the origination date (YYQn)
- **Property Details:**
    - **flag_fthb** (Categorical): Indicates whether Borrower(s) is a purchasing mortgaged property as a primary residence, having had no ownership interest in a residential property in the three years prior: Yes(Y), No(N), Not available/applicable(9)
    - **cd_msa** (Categorical): Metropolitan Statistical Area (MSA) or Metropolitan Division
    - **cnt_units** (Ordinal)/(Categorical) Denotes number of units in property: One(1), Two(2), Three(3), Four(4), Not available(99)
    - **occpy_sts** (Categorical): Indicates status of property: Owner occupied(P), Second home(S), Investment property(I), Not available (9)
    - **st** (Categorical): Indicates State to which property belongs
    - **prop_type** (Categorical): Denotes property type: Condominium(CO), Planned unit development(PU), Cooperative share(CP), Manufactured home(MH), Single-family home(SF)
    - **zipcode** (Categorical): Postal code for location of mortgaged property
    - **property_val** (Categorical): Indicates method used to obtain property appraisal, if any: ACE Loans(1), Full Appraisal(2), Other Appraisals(3), ACE+PDR(4), Not Available(9)

The following columns from the data set have been removed since they do not offer any additional information: - **ppmt_pnlty**, **io_ind** and **prod_type** are categorical features that only contain one value - **id_loan_rr** can be removed since it is an indicator that is also given by **rr_ind**.

[6]:
```
""" LOADING IN THE DATA """
d = pd.read_csv("freddiemac_extra2.csv", low_memory=False)
d = d.drop(columns=['id_loan_rr', 'io_ind', 'prod_type', 'ppmt_pnlty'])
```

2.1 Missing Data

The data includes approximately 20,000 null values for the `cd_msa` information. Despite this, there is no missing information on `zipcode`. As a result, we have decided to use zipcode data in analysis. This will be discussed in more detail in feature engineering. The remaining features that contain null values are `flag_sc`, and `rr_ind`, however these are categorical variables that take NaN to refer to 'No'. All NaN entries have been replaced with 'N'.

There are 41 missing Credit Scores in the dataset. Out of these, 3 loans were defaults. An option here could be to discretise the information, however the missing values represent Credit Scores less than 350 or greater than 750, which makes it hard to distinguish which extreme they are on. Furthermore, the `fico` column has a vast range of values, therefore discretising would lose some of this granularity. We proceed by removing these observations from the dataset. Only one observation is not available for `mi_pct` or mortgage insurance percentage and given this loan was not a default, we will exclude it from the data set. There are 6 missing observations for `cltv`, and 2 of these are the missing values for `ltv`, which occurs when the borrower has no other loans. None of these loans were defaults, so they have been excluded.

There are 2,412 missing values for `dti`, or the debt-to-income ratio, More notably, a disproportonate number of these observations are loans that have defaulted. When debt-to-income ratio is greater than 65%, it is classified as a missing value. These are loans where the monthly debt payments are greater than 65% of monthly income of the loanee suggesting that they are higher-risk loans. To deal with this, we have discretised debt-to-income and encoded it ordinally. We have used cross-validation to determine a reasonable choice for the number of bins as 10. These changes are made in pre-processing.

We interpret the value of 9 for `program_ind` as Not Applicable meaning that the loan is not part of a program. There could be missing data embedded within this, however since this category acts as a baseline associated to no programs, then this assumption does not lose any information. Finally, there are 125 values missing for `property_val`. None of these are defaults, so they have been excluded.

[7]:
```
""" REMOVING MISSING VALUES """
d['flag_sc'] = d['flag_sc'].fillna('N')
d['rr_ind'] = d['rr_ind'].fillna('N')
d = d[d['fico'] != 9999]
d = d[d['mi_pct'] != 999]
d = d[d['cltv'] != 999]
d = d[d['property_val'] != 9]
```

2.2 Feature Engineering

Date value changes There are 3 instances of date values with the dataset: `dt_matr`, `dt_first_pi` and `id_loan`. For `dt_matr` and `dt_first_pi` we convert the dates to current year plus the fraction of how much of that year has progressed (e.g. 03/2024 -> 2024.25). For `id_loan` we have the year and the fiscal quarter, where the dates are in the same end format above with the month being the mid-point of the quarter (e.g. Q1 is (January to March) hence the month assigned is February). These feature engineering steps ensure that these dates are suitable as numeric features in our model.

[4]:
```
# Converting Date Strings to Numeric
date_str_to_num = np.vectorize(lambda x : int(str(x)[:-2])+(int(str(x)[-2:])-0.
↪5)/12)
date_strq_to_num = np.vectorize(lambda x : 2000+int(x[1:3])+(int(x[4:5])*3-1)/
↪12)

d["dt_first_pi"] = date_str_to_num(d["dt_first_pi"])
d["dt_matr"] = date_str_to_num(d["dt_matr"])
```

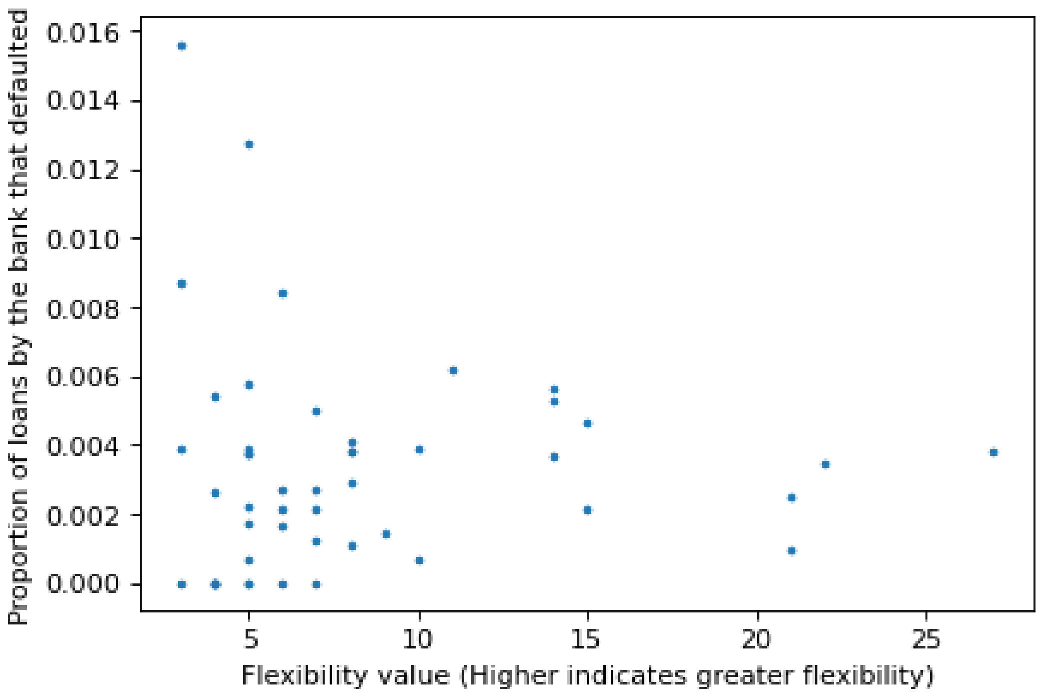
```
d["dt_logged"] = date_strq_to_num(d["id_loan"])
```

Banks One issue arises with the two categorical variables: `seller_name` and `servicer_name`. Both have many values, and if one-hot encoded, would add a lot of sparse data attributes. Hence, we created a new metric to capture information about `seller_name` numerically (where we dropped `servicer_name`). This metric is mortgage "flexibility", the number of unique mortgage lengths for each seller, with the thought process being, "flexible" banks are likely to be bigger banks with better risk assessment.

This decision is motivated by the data. The plot below demonstrates clearly that banks with a low "flexibility" rating tend to have higher default rates, while more flexible banks have consistently low defaulting proportion.

[9]:
```
""" Feature Engineering Step for bank flexbility """

bank_array = []
for bank in d["seller_name"].unique():
    if bank == "Other sellers":
        continue
    bank_data = d[d["seller_name"] == bank]
    # calculate flexibility
    flex = len(bank_data["orig_loan_term"].unique())
    default_prop = sum(bank_data["loan_status"]=="default")/
 ↪len(bank_data["loan_status"])
    bank_array.append([bank, flex, default_prop])

bank_df = pd.DataFrame(bank_array, columns=["seller_name", "flex",␣
 ↪"default_prop"])
other_banks = d[d["seller_name"] == "Other sellers"]
bank_df.loc[bank_df.shape[1]] = ["Other sellers", int(np.mean(bank_df["flex"])),
                                 sum(other_banks["loan_status"]=="default")/
 ↪len(other_banks["loan_status"])]
flex_vector = []
for bank in d["seller_name"]:
    if bank == "UNITED SHORE FINANCIAL SERVICES, LLC, DBA UNITED WHOLESALE M":
        flex_vector.append(0.00786)
        continue
    flex_vector.append(bank_df[bank_df["seller_name"] == str(bank)]["flex"].
 ↪values[0])
d["flex"] = flex_vector
```

[6]:
```
fig, ax = plt.subplots(figsize=(6,4))
ax.scatter(bank_df["flex"], bank_df["default_prop"], s=5)
ax.set_xlabel("Flexibility value (Higher indicates greater flexibility)")
ax.set_ylabel("Proportion of loans by the bank that defaulted")
plt.show()
```



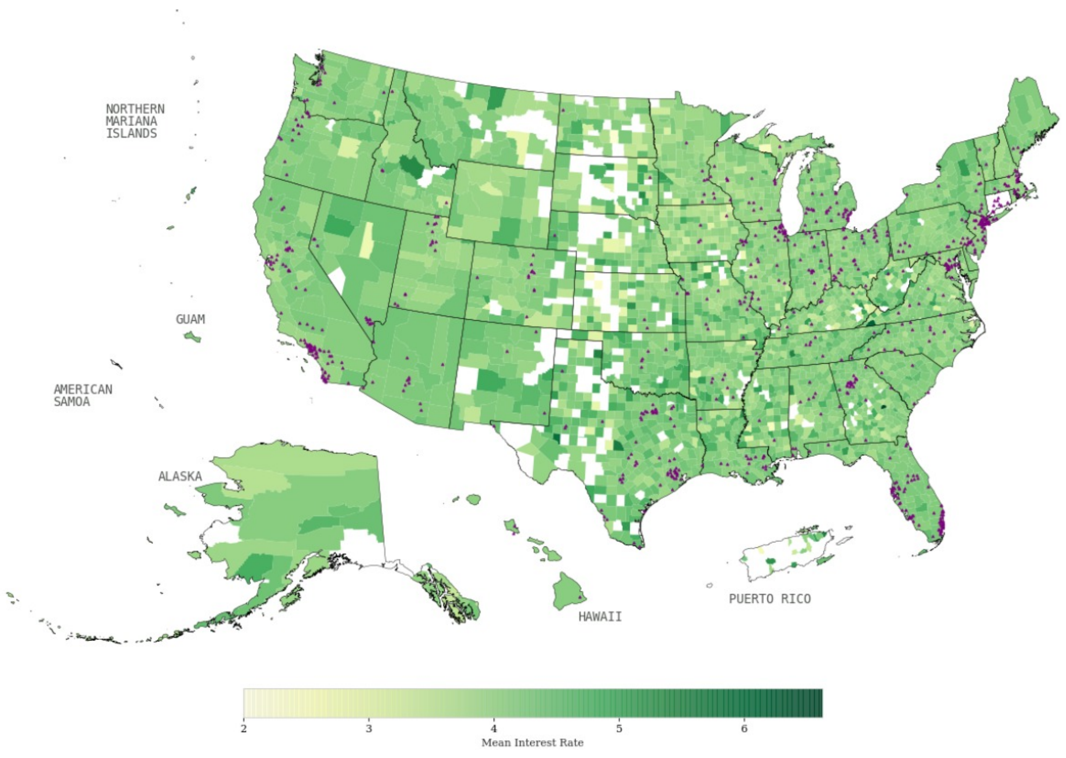
Geographical Data The main two geographical attributes within the dataset are `cd_msa` and `zipcode`. In there current form they are categorical, with a large number of categories (448 unique MSA codes and 886 unique zipcodes). Ideally, we would like to encode geographical location numerically.

For each zipcode entry, we have the first 3 digits but the last two are excluded for privacy. We have sourced a dataset externally, which indicates the latitude, longitude and population of different zipcodes. For each entry in the dataset, we took the beginning zipcode and assigned the final two digits based on a random sample weighted by proportion of zipcode population. Then, with each sample having a unique, 6 digit zipcode, we assigned the observation the latitude and longitude value for this zipcode. This was carried out before splitting active and prepaid/default loans.

The plot below highlights differing mean interests across zipcodes in the dataset. Defaults seem to appear in clusters of cities, where we would expect more loans be taken out. We have written a function to plot this using geopandas. The link to some reference code is found in references.

[7]:
```
from IPython.display import Image
Image('png/INT_RT.jpg')
```

[7]:



2.2.1 Data Split

To avoid any data leakage, the dataset is split into training and test sets from this point onwards.

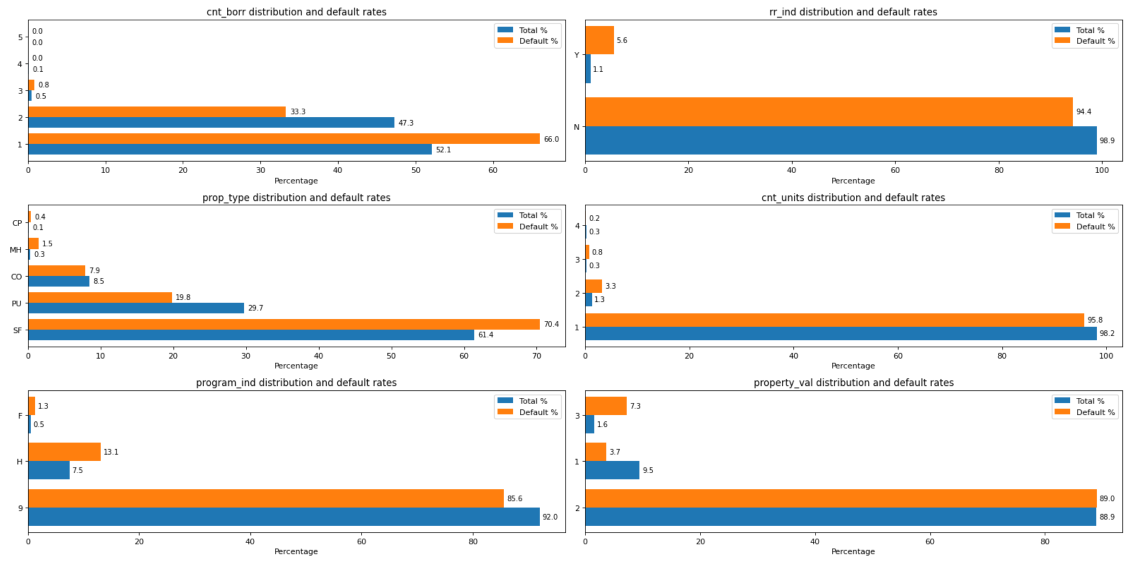
We will separate active loans from non-active (prepaid/defaulted) loans. The non-active loans will be separated into a training set (70%) to build the model, a validation set to tune the model (15%), and finally a test set (15%). Once the final model has been determined, it will be used to offer insight on existing loans with Freddie Mac.

[10]:
```
active = d[d['loan_status'] == 'active']
nonactive = d[d['loan_status'] != 'active']
nonactive = nonactive[[col for col in nonactive.columns if col !=␣
 ↪'loan_status'] + ['loan_status']]
# Feature matrix and response vector
X, y = nonactive.drop(['loan_status'], axis=1), nonactive['loan_status']
# Convert to numpy array
X = X.values

# Encode default
y = LabelEncoder().fit_transform(y)

# Naively split the data into train and test sets
X_train, X_tv, y_train, y_tv = train_test_split(X, y, shuffle= True,
                                                    test_size = 0.3,␣
 ↪random_state=1112, stratify=y)

X_test, X_val, y_test, y_val = train_test_split(X_tv, y_tv, shuffle= True,
                                                    test_size = 0.5,␣
 ↪random_state=1112, stratify=y_tv)

# Convert back to DataFrame
df_train = pd.DataFrame(np.concatenate([X_train, y_train.reshape(-1, 1)],␣
 ↪axis=1), columns=nonactive.columns)
df_test = pd.DataFrame(np.concatenate([X_test, y_test.reshape(-1, 1)], axis=1),␣
 ↪columns=nonactive.columns)
df_val = pd.DataFrame(np.concatenate([X_test, y_test.reshape(-1, 1)], axis=1),␣
 ↪columns=nonactive.columns)
```

3 Exploratory Data Analysis

3.0.1 Categorical Data

[11]:
```
plot_categorical(df_train, ['cnt_borr', 'rr_ind', 'prop_type', 'cnt_units',␣
 ↪'program_ind', 'property_val'])
```



We have examined the distributions of categorical data alongside default rates within these categories to indicate which loans could be higher risk. The findings for each feature, based on the plots above, are outlined as follows:

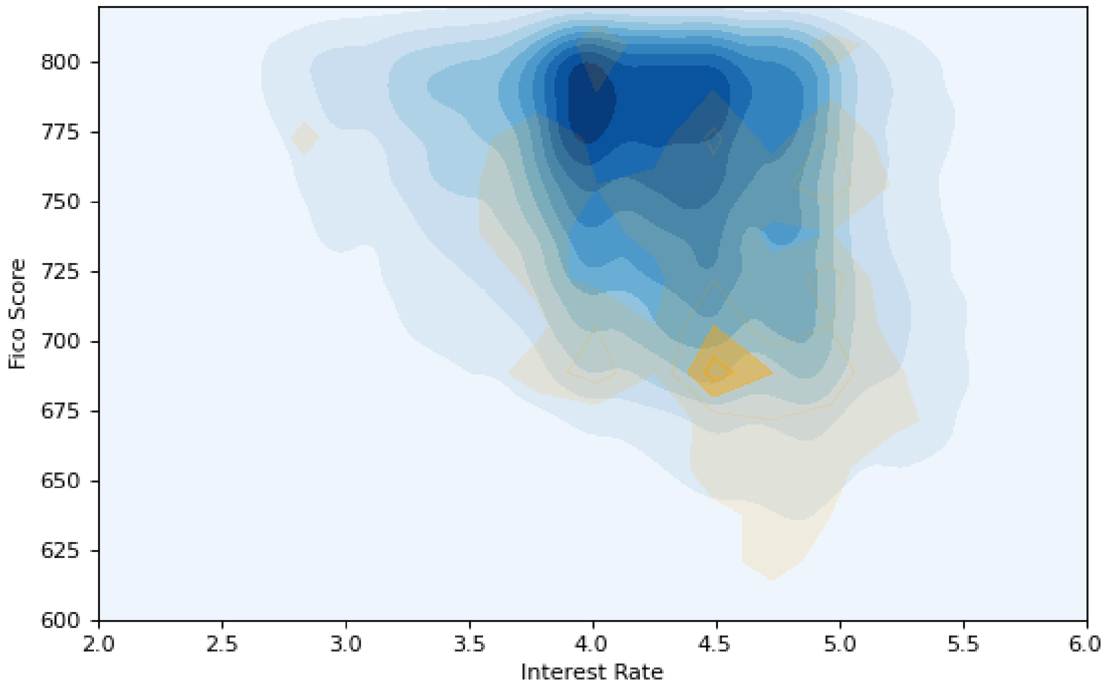
- `cnt_borr` : When there is one borrower on the loan, a default is much more likely, when compared with two borrowers. This follows expectations since there is more stability when financial responsibility is shared.

- `rr_ind` : Despite making up a small proportion of the dataset, loans that are on the Relief Refinance program are disproportionately more likely to default.
- `prop_type` : Manufactured Homes are more likley to default, whereas Planned Unit Developments have a lower default rate.
- `cnt_units` : Most mortgages cover properties with 1 unit, and therefore these observations make up most of the defaults. However, 2 unit properties have a disproportionate rate of defaults.
- `program_ind` : Mortages on the Home Possible and HFA Advantage programs have a higher rate of defaults than the proportion of the data that they comprise.
- `property_val` : Mortgages that have been received Other Appraisals have a disproportionately high number of defaults.

3.0.2 Numerical Data

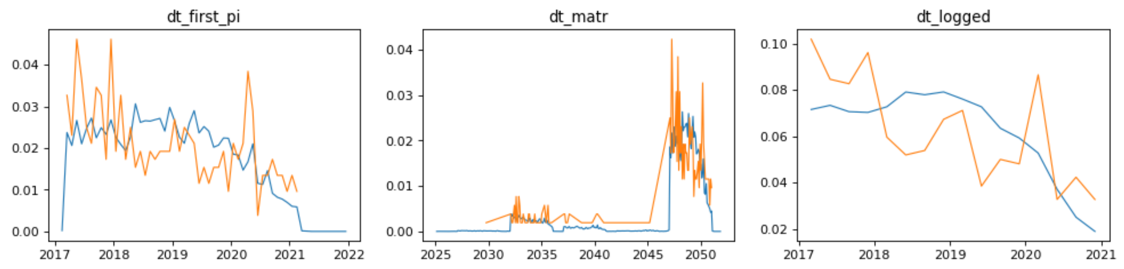
We check our assumptions for variables that would likely indicate someone to default or not, being the interest rate on their mortgage (int_rt) and fico score (fico). Looking at the heatmap below, we can see in blue the heatmap of the entirety of the training dataset, laid onto is a heatmap of those who have defaulted. Clearly we can see the orange/default heatmap is centered on a low fico score (in comparison to the whole population) and at a higher interest rate than the main proportion seen in the entire dataset.

[10]:
```
plot_heatmap(df_train, "int_rt", "fico")
```



[11]:
```
default_df = df_train[df_train["loan_status"]==0]
prepaid_df = df_train[df_train["loan_status"]==1]
```

```
fig, ax = plt.subplots(1,3, figsize = (15,3))
for i,var in enumerate(["dt_first_pi", "dt_matr", "dt_logged"]):
    dates_people_buy = np.unique(prepaid_df[var], return_counts=True)
    dates_people_buy_default = np.unique(default_df[var], return_counts=True)

    ax[i].plot(dates_people_buy[0], dates_people_buy[1]/
 ↪sum(dates_people_buy[1]),
                linewidth=1)
    ax[i].plot(dates_people_buy_default[0], dates_people_buy_default[1]/
 ↪sum(dates_people_buy_default[1]), linewidth=1)
    ax[i].set_title(var)
```



Exploring the tranformed date variables, the above plots display the proportion of samples on each date to the total number of samples. The blue line show for the whole training dataset, the orange shows the subset of those that defaulted on their mortgages. Clearly there is little helpful information with `dt_matr`, and the attributes `dt_first_pi` and `dt_logged` show the same trend, with `dt_logged` being smoother as it is in terms of quarters not months. We clearly see two high points on the orange (default line), one throughout the year of 2017 and another right at the start of 2020. The spike at the start of 2020 is likely due to the COVID-19 pandemic.

3.1 Pre-processing

[55]:
```
# Column Transformer Pipeline

dti_index = df_train.columns.get_loc("dti")
num_var =␣
 ↪["fico","dt_first_pi","dt_logged","flex","mi_pct","orig_upb","int_rt","dt_matr","cltv",␣
 ↪"ltv", "orig_loan_term", "lat", "lng"]
cat_var = ["flag_fthb", "cnt_units", "occpy_sts", "channel", "prop_type",␣
 ↪"loan_purpose", "cnt_borr", "flag_sc", "program_ind", "rr_ind",␣
 ↪"property_val", "mi_cancel_ind"]

# indexing for features
num_var_index = df_train.columns.get_indexer(num_var)
```

```
cat_var_index = df_train.columns.get_indexer(cat_var)

ct = Pipeline(
    [("pre_processing", ColumnTransformer(
        [("disc", KBinsDiscretizer(encode='ordinal', n_bins=10), [dti_index]),
         ("cat", OneHotEncoder(drop='first', handle_unknown='ignore'),␣
 ↪cat_var_index),
         ("num", StandardScaler(), num_var_index),
         ], remainder="drop",
    ))])

ct.fit(X_train)

# transform data
X_train_e = ct.transform(X_train)
X_val_e = ct.transform(X_val)
X_test_e = ct.transform(X_test)
cat_encoded =  ['flag_fthb_Y', 'cnt_units_2', 'cnt_units_3', 'cnt_units_4',␣
 ↪'occpy_sts_P', 'occpy_sts_S', 'channel_C', 'channel_R', 'prop_type_CP',␣
 ↪'prop_type_MH', 'prop_type_PU', 'prop_type_SF', 'loan_purpose_N',␣
 ↪'loan_purpose_P', 'cnt_borr_2', 'cnt_borr_3', 'cnt_borr_4', 'cnt_borr_5',␣
 ↪'flag_sc_Y', 'program_ind_F', 'program_ind_H', 'rr_ind_Y', 'property_val_2',␣
 ↪'property_val_3', 'mi_cancel_ind_N', 'mi_cancel_ind_Y']
feature_names = ['dti'] + cat_encoded + num_var
```

3.1.1 Dealing with Imbalance

Our task is a binary classifier and hence is highly sensitive to label imbalance. In the case of our dataset, this imbalance is vast. Particularly, we have very few instances of people who have defaulted. To combat this, we carry out resampling. There are two main methods, under- or oversampling; undersampling in our case is unfavorable as it removes a large quantity of observations for people who have prepaid their mortgages. Hence, we will be using oversampling.

Two main methods of oversampling are being considered:

- **Random oversampling** - where we resample from the subset of mortgages that default. Due to the size of the imbalance, these samples will have a large number of duplicates. This is not advantageous, as our model is likely to overfit due to the low variability in the subset of defaulting mortgages. (note: this can be mitigated by adjusting shrinkage, this will be explored).
- **SMOTE (*Synthetic Minority Oversampling Technique*)** - generates new synthetic observations via k nearest neighbours, randomly selects an observation. Finds the k nearest neighbours to this observation, chooses one at random and generates a new observation between itself and neighbour selected. This helps to avoid overfitting the data although will likely cause our dataset to misclassify those in the majority class (prepaid).

[56]:
```
# oversampling data
OS = RandomOverSampler(random_state=1112)
```

```
X_train_e_over, y_train_over = OS.fit_resample(X_train_e, y_train)
# smote data
SMTE = SMOTE(random_state=1112)
X_train_e_smote, y_train_smote = SMTE.fit_resample(X_train_e, y_train)
```

4 Model Fitting and Tuning

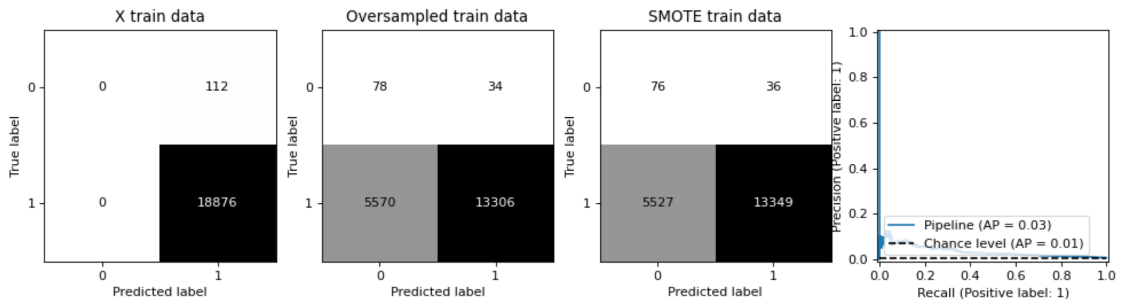
4.0.1 Logistic Regression - Baseline Model

The performance of the logistic regression model is poor with the relatively high recall rate of nearly 70% coming at the cost of extremely low precision of around 1.3%. This is made clear by the precision-recall curve, which shows performance which rapidly drops to the chance level. The confusion matrices below indicate the performance of a logistic regression model trained on the original training set, an oversampled training set and a SMOTE training set. The standard set further motivates the need to deal with imbalanced data. When the model is trained on the original set, the model treats identifying someone as non-default, when they default (false negative) is considered as negligible since there are so few data of this class. As a result, the default recall is extemely low in both training and validation sets. This issue is dealt with in the oversampling and SMOTE training sets, however in these cases, there is an extremely high false positives, which despite being less costly to the company, still want to be avoided. This poor performance is likely due to the linear assumption that is made by a Logistic Regression model. Since the data contains many non-linear trends, the model may have oversimplfied the separation of the data, making it sensitive to either hihger false positives or false negatives. Nevertheless, due to its simplicity and easy interpretabilty, we have chosen this model as a baseline to compare with our more developed models.

[57]:
```
# Logistic regression Pipeline
log_pipe = make_pipeline(
    LogisticRegression(random_state=42, penalty=None)
)
fig, ax = plt.subplots(1,4, figsize = (15,5))
# Fit the model using the training data
log_orig = log_pipe.fit(X_train_e, y_train)
ConfusionMatrixDisplay.from_estimator(log_orig, X_val_e, y_val,
                                      ax=ax[0], cmap="binary", colorbar=False)
ax[0].set_title("X train data")

# Fit the model using the over-sampled training data
log_over = log_pipe.fit(X_train_e_over, y_train_over)
ConfusionMatrixDisplay.from_estimator(log_over, X_val_e, y_val,
                                      ax=ax[1], cmap="binary", colorbar=False)
ax[1].set_title("Oversampled train data")

# Fit the model using the SMOTE training data
log_smote = log_pipe.fit(X_train_e_smote, y_train_smote)
ConfusionMatrixDisplay.from_estimator(log_smote, X_val_e, y_val,
                                      ax=ax[2], cmap="binary", colorbar=False)
```

```
ax[2].set_title("SMOTE train data")

model_inverted = log_pipe.fit(X_test_e,1- y_test)
PrecisionRecallDisplay.from_estimator(model_inverted, X_test_e, 1- y_test,␣
  ↪plot_chance_level=True, ax=ax[3])
plt.show()
```

```
/Users/alfie/anaconda3/lib/python3.12/site-
packages/sklearn/linear_model/_logistic.py:469: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. OF ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
/Users/alfie/anaconda3/lib/python3.12/site-
packages/sklearn/linear_model/_logistic.py:469: ConvergenceWarning: lbfgs failed
to converge (status=1):
STOP: TOTAL NO. OF ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
```

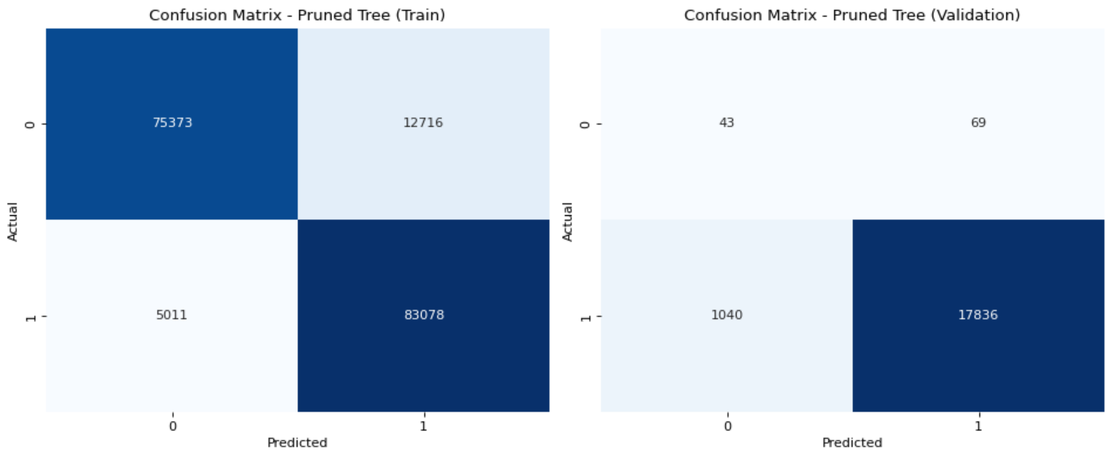


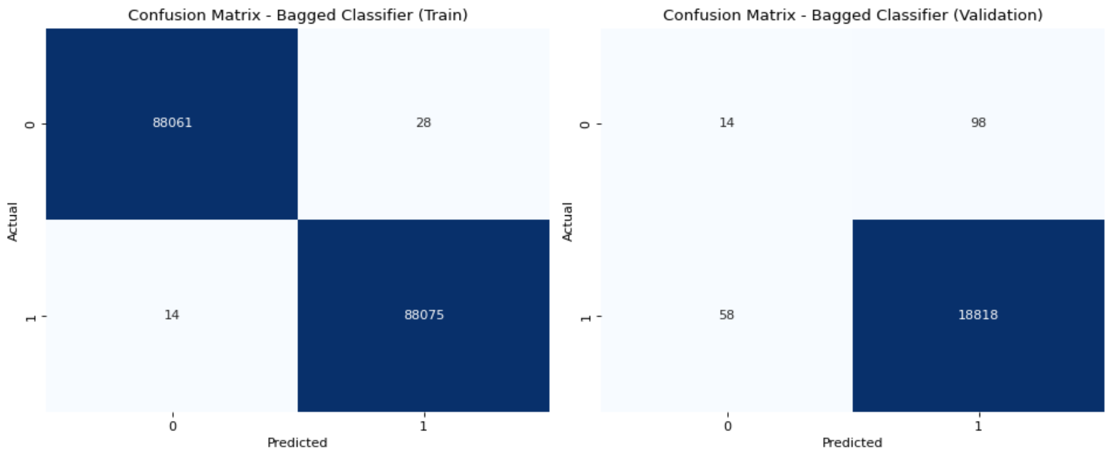
4.0.2 Support Vector Machines

Due to the size of the dataset, the training time for a Support Vector Machine model is significantly high. To overcome this, we have considered a few strategies. One option is to reduce the feature space. Using the results from our feature importance model, which is discussed later, some of the least important features were removed, however this did not reduce the training time much. To reduce the dataset, we could also train the model on a subset of the training set. When this is done using an *rbf* kernel using a 50% sub-sample, the training time is approximately 5 minutes. The training time is more reasonable for a linear kernel, however, this model suffers with the same issues as the logisitic regression model, so we have chosen to exclude this model.

4.0.3 Random Forests

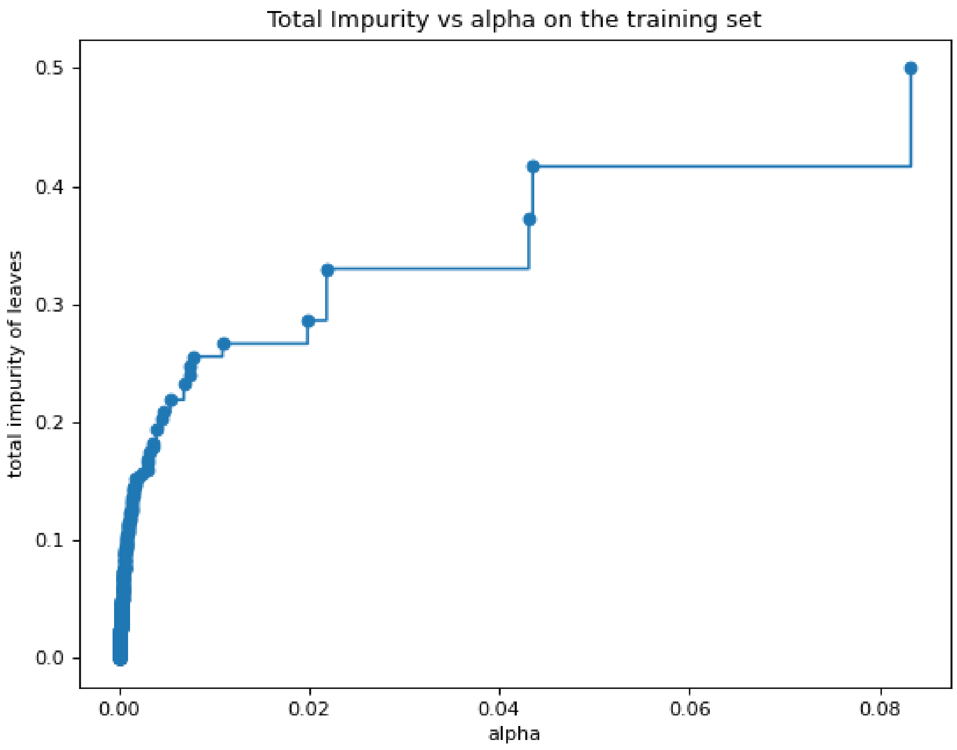
```
[120]: display(Image('png/Pruned_Tree.png',width=600))
       display(Image('png/Bagging.png',width=600))
```

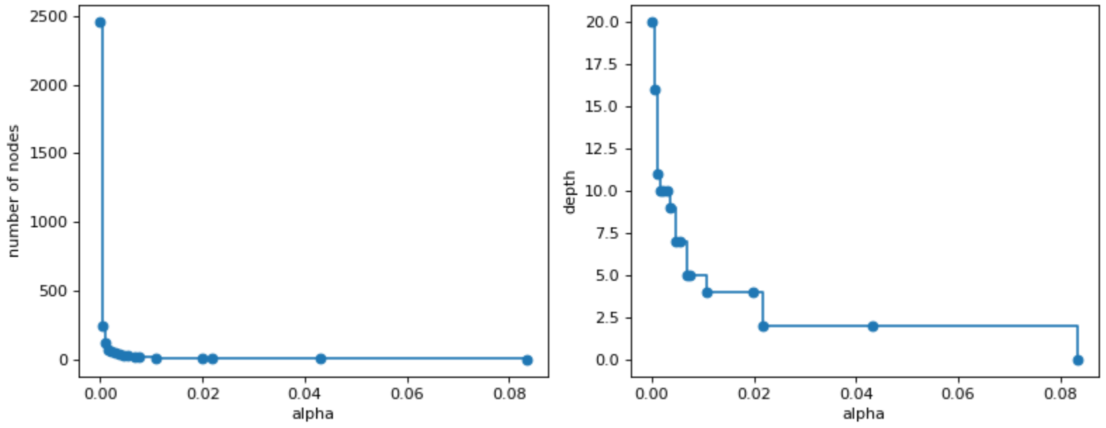




The categorical nature of this task lent itself to the use of decision trees. We investigated Pruned Trees, Bagging and Random Forests in order to find the most effective method for our problem. We first considered Pruned Trees, for which we tuned our alpha value with a combination of cross-validation and plotting the metrics of total impurity, number of nodes, tree-depth and accuracy against different alpha values. From this we selected a value of $\alpha = 0.002$ which resulted in a Pruned Tree with a depth of 10 and 61 nodes. The pruned tree performed fairly well, correctly identifying 43 defaulters in our validation set, representing 38.4% of the total defaulters in the set. There were 1040 false positives, meaning 1040 non-defaulters were incorrectly classified as defaulters. This accounts for 5.5% of the non-defaulters in the validation set.

[132]:
```
display(Image('png/Impurity.png',width=50))
display(Image('png/Nodes&Depth.png',width=500))
```





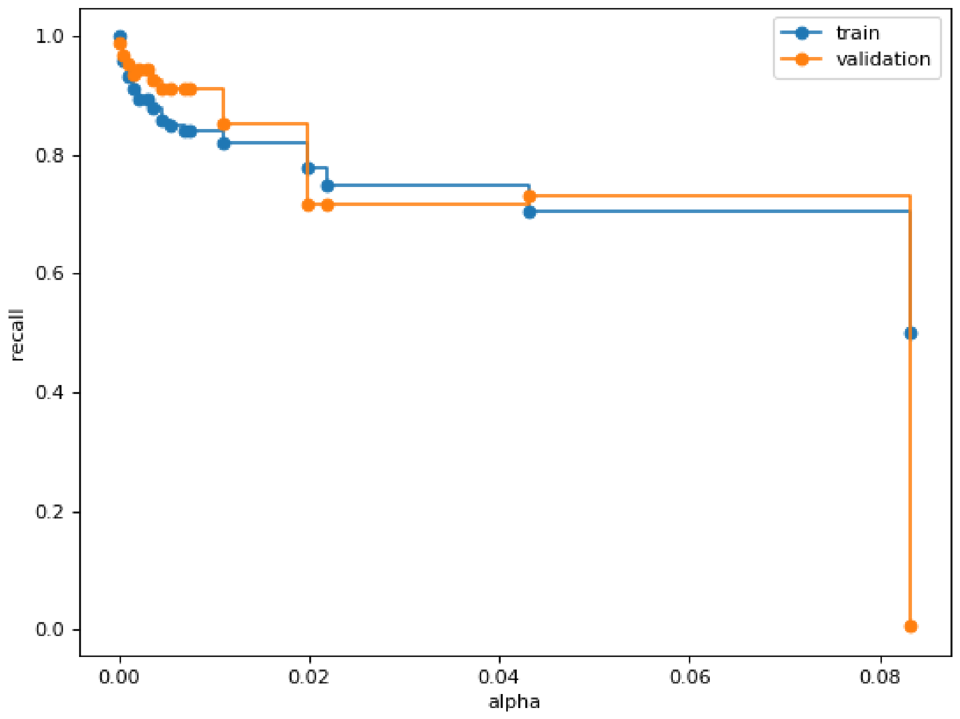
Whilst bagging performed extremely well on the training data, only misclassifying 0.0002% of the data points in our training set, it performed comparatively poorly on the validation set. It only correctly identified 14 defaults, 12.5% of defaults in the validation set. However, it also maintained a very low number of false positives at 58, representing 0.003% of all defaulters.

Finally, we tuned a Random Forest model with parameters `n_estimators = 100` and `max_depth = 3`. The Random Forest model correctly identified 59 defaults, representing 52.5% of defaults in the validation set. The trade-off for this was a much higher number of false-positives, at 3958. This accounted for 21% of all of the non-defaulters.

When comparing the three models, bagging clearly falls short of the two other options in this context. While its low false positive rate increases confidence in the predictions it does make, the model ultimately fails to be effective enough at detecting defaults to be considered effective.

One reason for the apparent overfitting of the Bagging method could be due to the fact that, with such a small set of defaulters in the original dataset, the synthetic data points generated by the SMOTE may not reflect the true distribution of such data points in the validation set. Thus the model may be learning to recognise synthetic characteristics in place of the actual minority patterns. This means that bagging can create similar base learners which amplify the same noise of the minority set, undermining its performance on unseen data. Random Forests, in contrast, are specifically designed to introduce variation among base learners through random feature selection, making it better suited to handle such issues.

```
[135]: display(Image('png/Recall.png', width=50))
```



We have decided to use the Random Forest model over the Pruned Tree model. Whilst, the Pruned Tree model identifies significantly less false positive that the Random Forest Model, we feel that a model that fails to correctly identify 40% of its target, cannot be used over one that managed this. Furthermore, we stress the importance of identifying potential defaulters due to the large costs the incur on the bank.

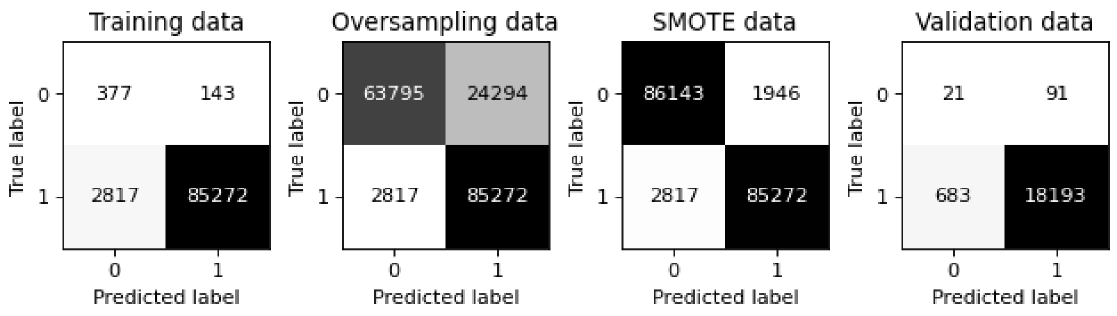
[58]:
```
"""Final Random Forest Model after tuning"""
rf_model = RandomForestClassifier(max_depth=3, random_state=1112,␣
 ↪n_estimators=100).fit(X_train_e_smote, y_train_smote)
```

4.0.4 Neural Networks

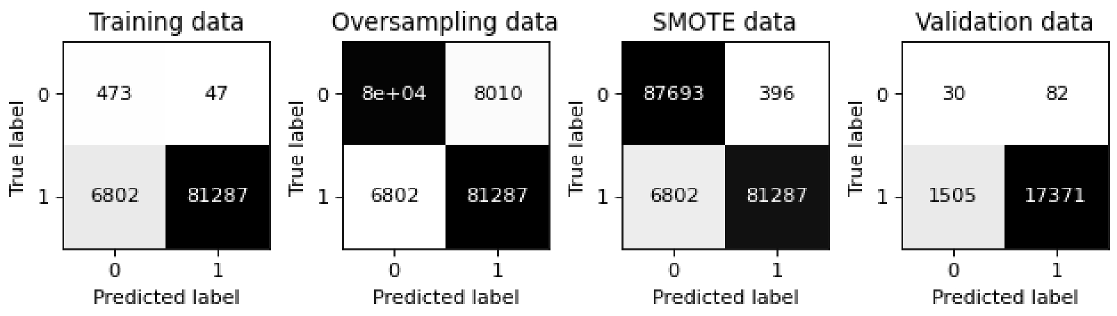
We now consider neural network models. First starting with a simple baseline,

[26]:
```
input_shape = int(X_train_e.shape[1])

# baseline neural network
nn_model_base = keras.models.Sequential([
    keras.layers.Input((input_shape,)),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(1, activation='sigmoid')
])
nn_model_base.compile(loss='binary_crossentropy', metrics=['accuracy'])
history_base = nn_model_base.fit(X_train_e_smote, y_train_smote, epochs=10,␣
 ↪validation_split=0.2, verbose=0)
```

[27]:
```
data_pairs = [(X_train_e, y_train),
              (X_train_e_over, y_train_over),
              (X_train_e_smote, y_train_smote),
              (X_val_e, y_val)]

data_pair_labels = ["Training data", "Oversampling data", "SMOTE data",␣
 ↪"Validation data"]

produce_scoring(nn_model_base, data_pairs, data_pair_labels, threshold=0.5)
```



[28]:
```
produce_scoring(nn_model_base, data_pairs, data_pair_labels, threshold=0.8)
```

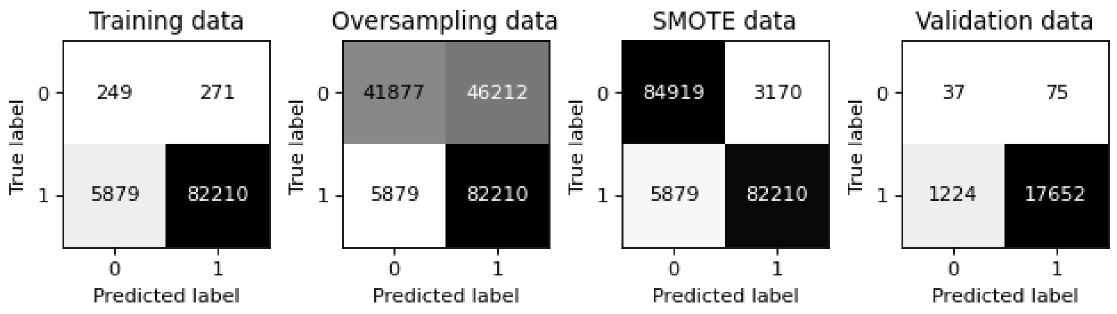


We can see based on the above that with a higher threshold for choosing the binary label from the probabilities provides better recall for observations within the default class. This seems to suggest saturation within the neural network. Quickly checking the distributions of the attributes of X_train_e_smote we can see why this is likely, with lots of values within the numerical data columns lie far outside the interval of 0 to 1 (as our output is between 0 and 1 this is problematic). Hence, we carry out min-max scaling.

```
[29]: # Fitting scaler.
min_max_scaler = MinMaxScaler()
min_max_scaler.fit(X_train_e)

# Transforming our datasets.
X_train_e_nn = min_max_scaler.transform(X_train_e)
X_val_e_nn = min_max_scaler.transform(X_val_e)
X_train_e_smote_nn = min_max_scaler.transform(X_train_e_smote)
X_train_e_over_nn = min_max_scaler.transform(X_train_e_over)

data_pairs_nn = [(X_train_e_nn, y_train),
               (X_train_e_over_nn, y_train_over),
               (X_train_e_smote_nn, y_train_smote),
               (X_val_e_nn, y_val)]
```

```
[31]: nn_model_base_mm = keras.models.Sequential([
    keras.layers.Input((input_shape,)),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(1, activation='sigmoid')
])

nn_model_base_mm.compile(loss='binary_crossentropy', metrics=['accuracy'])
history_base_mm = nn_model_base_mm.fit(X_train_e_smote_nn, y_train_smote,␣
  ↪epochs=10, validation_split=0.2, verbose=0)
```

```
[ ]: produce_scoring(nn_model_base_mm, data_pairs_nn, data_pair_labels, threshold=0.
  ↪5)
```



We see improvements on the validation set although this model now performs worse on the SMOTE set used to train. In order to further mitigate saturation we carry out the following:

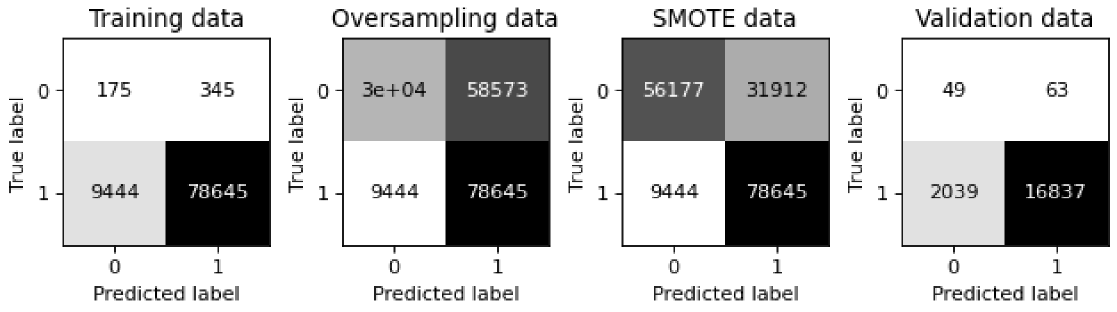
- **Batch Normalisation** (for each layer) - As our input and output values are within the range of 0 to 1, ideally our weights should be restricted to this interval.
- **He Normal initialization** - This set the initial values of each layer of weights to normal in distribution with some fixed variance (in the instance of this initialisation it is dependant on the number of neurons in the layer before the set of weights), this helps to mitigage exploding

gradients and staturation.

- **Elu activation function** - This activation function is similar to standard ReLu although instead of flattening out to 0 when x is negative, it returns a negative value (closer to -1 as x tends to negative infinity), this helps to keep the layer values in a distribution with mean 0.

[32]:
```
# adding normalisation
nn_model_norm = keras.models.Sequential([
    keras.layers.Input((input_shape,)),
    keras.layers.BatchNormalization(),
    keras.layers.Activation('elu'),
    keras.layers.Dense(100, activation=None, kernel_initializer='he_normal'),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(1, activation='sigmoid')
])

nn_model_norm.compile(loss='binary_crossentropy', metrics=['accuracy'])
history_norm = nn_model_norm.fit(X_train_e_smote_nn, y_train_smote, epochs=10,␣
 ↪validation_split=0.2, verbose=0)
```

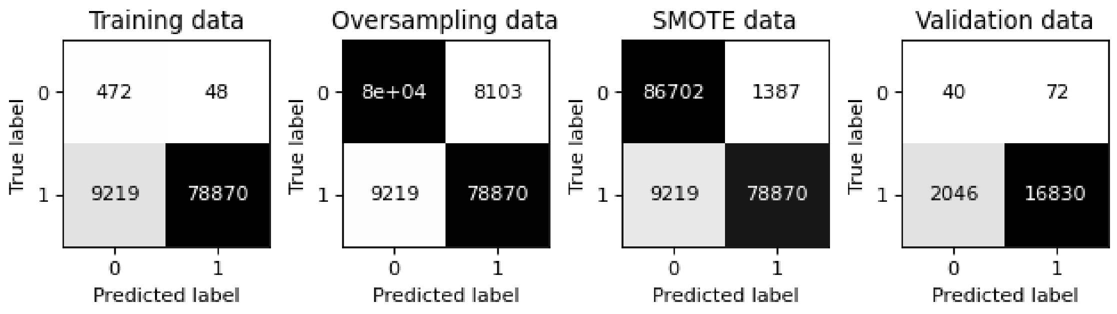
[33]:
```
produce_scoring(nn_model_norm, data_pairs_nn, data_pair_labels, threshold=0.5)
```



Our results are still not great although our neural network is still rather shallow, hence finally we look to add more depth to hopefully identify complex patterns in the data. Alongside this change we introduce dropout, as we are adding so many more weights it would be wise to ensure the model doesn't rely on a particular subset. One final change is to our loss function and optimizer. A binary focal cross entropy loss function is chosen to help emphasize the end criterion, we want to maximise default recall (it would not be advantageous to miss predict those who would default). Adam optimizer is chosen as we seek to converge quickly to mitigate the extra time taken to fit the extra layers.

[34]:
```
nn_model_deep = keras.models.Sequential([
    keras.layers.Input((input_shape,)),
    keras.layers.BatchNormalization(),
    keras.layers.Activation('elu'),
    keras.layers.Dense(100, activation=None, kernel_initializer='he_normal'),
    keras.layers.Dropout(0.2),
    keras.layers.BatchNormalization(),
    keras.layers.Activation('elu'),
    keras.layers.Dense(300, activation=None, kernel_initializer='he_normal'),
    keras.layers.Dropout(0.4),
    keras.layers.BatchNormalization(),
    keras.layers.Activation('elu'),
    keras.layers.Dense(300, activation=None, kernel_initializer='he_normal'),
    keras.layers.Dropout(0.4),
    keras.layers.BatchNormalization(),
    keras.layers.Activation('elu'),
    keras.layers.Dense(100, activation=None, kernel_initializer='he_normal'),
    keras.layers.Dropout(0.2),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(1, activation='sigmoid')
])

loss_func = keras.losses.BinaryFocalCrossentropy(alpha=0.55, label_smoothing=0.
  ↪05)
nn_model_deep.compile(loss=loss_func, optimizer="adam", metrics=['accuracy'])
history_deep = nn_model_deep.fit(X_train_e_smote_nn, y_train_smote, epochs=20,␣
  ↪validation_split=0.3, verbose=0)
```

```
[35]: produce_scoring(nn_model_deep, data_pairs_nn, data_pair_labels, threshold=0.5)
```



4.0.5 Final Model

After fine-tuning both the Neural Network model and Random Forest model, we can now compare the two. Before we do so, we wanted to see if there was similarity in their predictions, which would motivate an ensemble. Below is the cosine similarity between the two predicted probability vectors for the random forest model and the neural network model. This similarity is very close to one and hence suggests that each of our models is correctly identifing similar observations. From this we choose not to proceed with an ensemble of the two models and rather access their performance (along with the baseline logistic regression model).

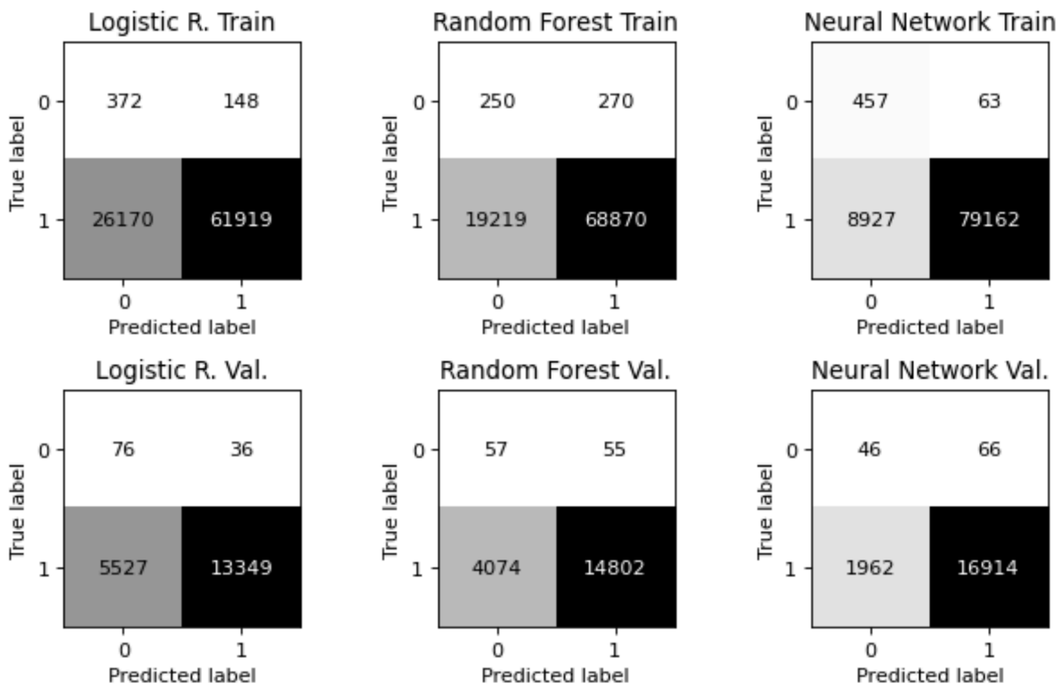
```
[59]: # Neural Network predictions
      y_pred_nn = nn_model_deep.predict(X_val_e_nn)
      # Random Forest predictions
      y_pred_rf = rf_model.predict_proba(X_val_e)
      print((y_pred_nn[:, 0]).dot(y_pred_rf[:, 0]) / (np.linalg.norm(y_pred_nn[:, 0])␣
       ↪* np.linalg.norm(y_pred_rf[:, 0])))
```

```
594/594              1s 1ms/step
0.8928434740882284
```

```
[ ]: t = 0.5

     # convert probabilities to labels
     to_binary = np.vectorize(lambda x : 1 if x > t else 0)

     y_pred_log_train = log_smote.predict(X_train_e)
     y_pred_log_val = log_smote.predict(X_val_e)

     y_pred_rf_train = to_binary(rf_model.predict_proba(X_train_e)[:,1])
     y_pred_rf_val = to_binary(rf_model.predict_proba(X_val_e)[:,1])

     y_pred_nn_train = to_binary(nn_model_deep.predict(X_train_e_nn))
     y_pred_nn_val = to_binary(nn_model_deep.predict(X_val_e_nn))

     true_labels = [y_train, y_val]


     Image('png/cm.png', width=600)
```

```
2770/2770              3s 1ms/step
594/594              1s 1ms/step
```

```
[ ]:
```



Firstly, looking at the confusion matrices for each of the models, we can see that the Neural Network massively outperforms both the Logistic Regression model and the Random Forest model when it comes to separating the training data. When we look at the validation set (bottom row), the logistic baseline is correctly identifying the most defaults, however this comes at a huge cost of many false positives. The random forest model has slightly more balance here, but there is still a large portion of the dataset, which is being incorrectly labelled as a default. Using this model would mean that many loans are turned down, even though they are not high risk, resulting in a large oppourtunity cost. The neural network manages to balance this the best, with a reasonbable identification of defaults, alongside a very low false positive rate.

[41]:
```
metrics = {
    "Recall": (
        recall_score(y_val, y_pred_log_val),
        recall_score(y_val, y_pred_rf_val),
        recall_score(y_val, y_pred_nn_val),
    ),
    "F1 Score": (
        f1_score(y_val, y_pred_log_val),
        f1_score(y_val, y_pred_rf_val),
        f1_score(y_val, y_pred_nn_val),
    ),
    "Accuracy": (
        accuracy_score(y_val, y_pred_log_val),
        accuracy_score(y_val, y_pred_rf_val),
        accuracy_score(y_val, y_pred_nn_val),
    )
}

print("Metric        | Log Regression | Random Forest | Neural Network")
print("----------------------------------------------")
for metric, (lr_val, rf_val, nn_val) in metrics.items():
    print(f"{metric:<13} |  {lr_val:.4f}        | {rf_val:.4f}        | {nn_val:
↪.4f}")
```

```
Metric        | Log Regression | Random Forest | Neural Network
----------------------------------------------
Recall        |  0.0000        | 0.7842        | 0.8916
F1 Score      |  0.0000        | 0.8776        | 0.9408
Accuracy      |  0.0059        | 0.7825        | 0.8885
```

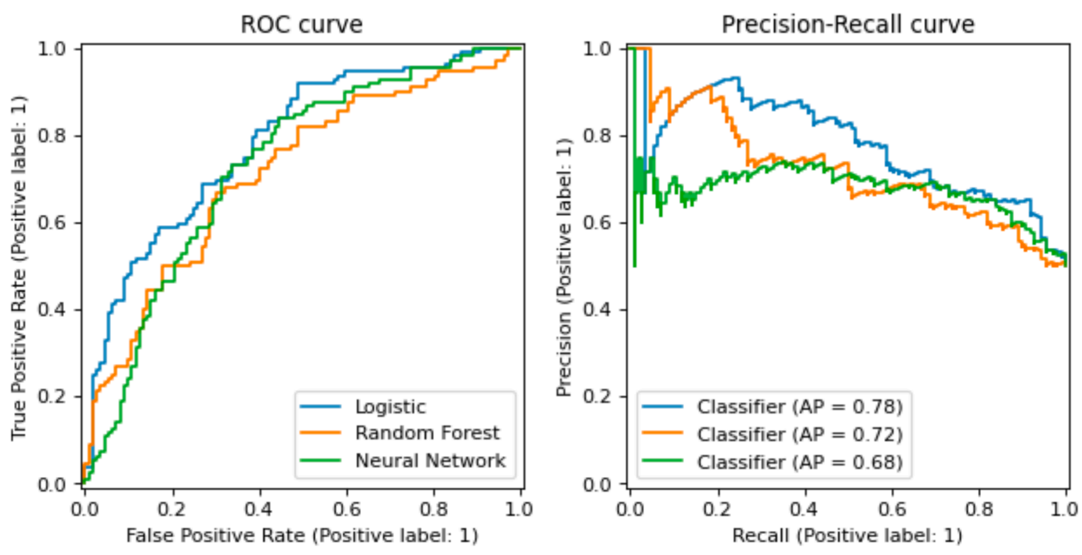
The Neural Network outperforms the other models across all three metrics:

- **Recall**: The Neural Network correctly identifies nearly 90% of all actual positive cases, which is significantly higher than both Logistic Regression and Random Forest.
- **F1 Score**: The highest F1 score of 94% for the Neural Network highlights its excellent balance between precision and recall, which is important because the firm needs to catch defaults, while balancing the priority of not flagging too many loans as defaults, when they are not.
- **Accuracy**: With an accuracy of almost 90%, the Neural Network demonstrates the best overall classification performance.

[124]:
```
""" ROC and Precision-Recall Curves"""

under_samp = RandomUnderSampler(random_state=1112)
X_val_e_even, y_val_even = under_samp.fit_resample(X_val_e, y_val)
X_val_e_even_nn = min_max_scaler.transform(X_val_e_even)

y_pred_log_val_even = log_smote.predict_proba(X_val_e_even)[:,1]
y_pred_rf_val_even = rf_model.predict_proba(X_val_e_even)[:,1]
y_pred_nn_val_even = nn_model_deep.predict(X_val_e_even_nn)

Image('png/roc_rp.png', width=600)
```

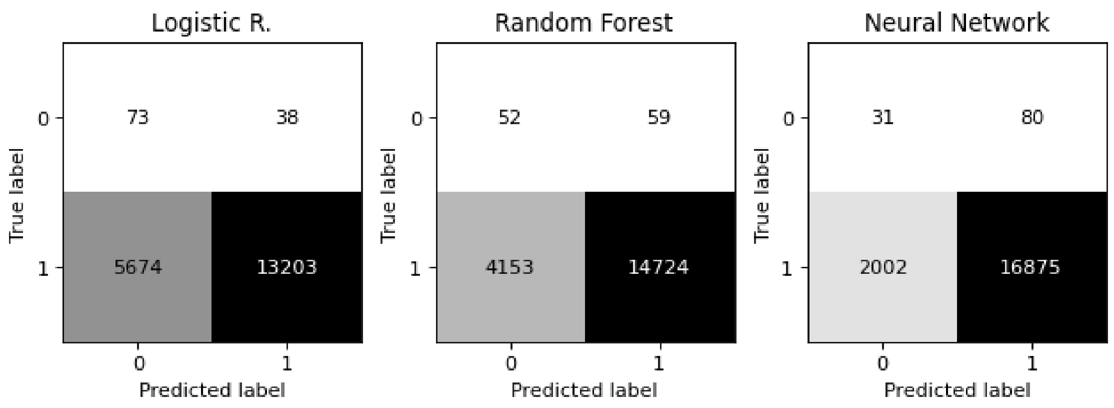
```
7/7              0s 4ms/step
```

[124]:



```
[ ]: """ Test Data Confusion Matrices"""

     X_test_e = ct.transform(X_test)
     X_test_e_nn = min_max_scaler.transform(X_test_e)

     y_pred_log_test = log_smote.predict(X_test_e)
     y_pred_rf_test = to_binary(rf_model.predict_proba(X_test_e)[:,1])
     y_pred_nn_test = to_binary(nn_model_deep.predict(X_test_e_nn))

     fig, ax = plt.subplots(1,3)

     ConfusionMatrixDisplay.from_predictions(y_test, y_pred_log_test,
                                             ax=ax[0], colorbar=False, cmap="binary")
     ConfusionMatrixDisplay.from_predictions(y_test, y_pred_rf_test,
                                             ax=ax[1], colorbar=False, cmap="binary")
     ConfusionMatrixDisplay.from_predictions(y_test, y_pred_nn_test,
                                             ax=ax[2], colorbar=False, cmap="binary")

     ax[0].set_title('Logistic R.'); ax[1].set_title('Random Forest'); ax[2].
      ↪set_title('Neural Network')
     plt.tight_layout()
     plt.show()
```

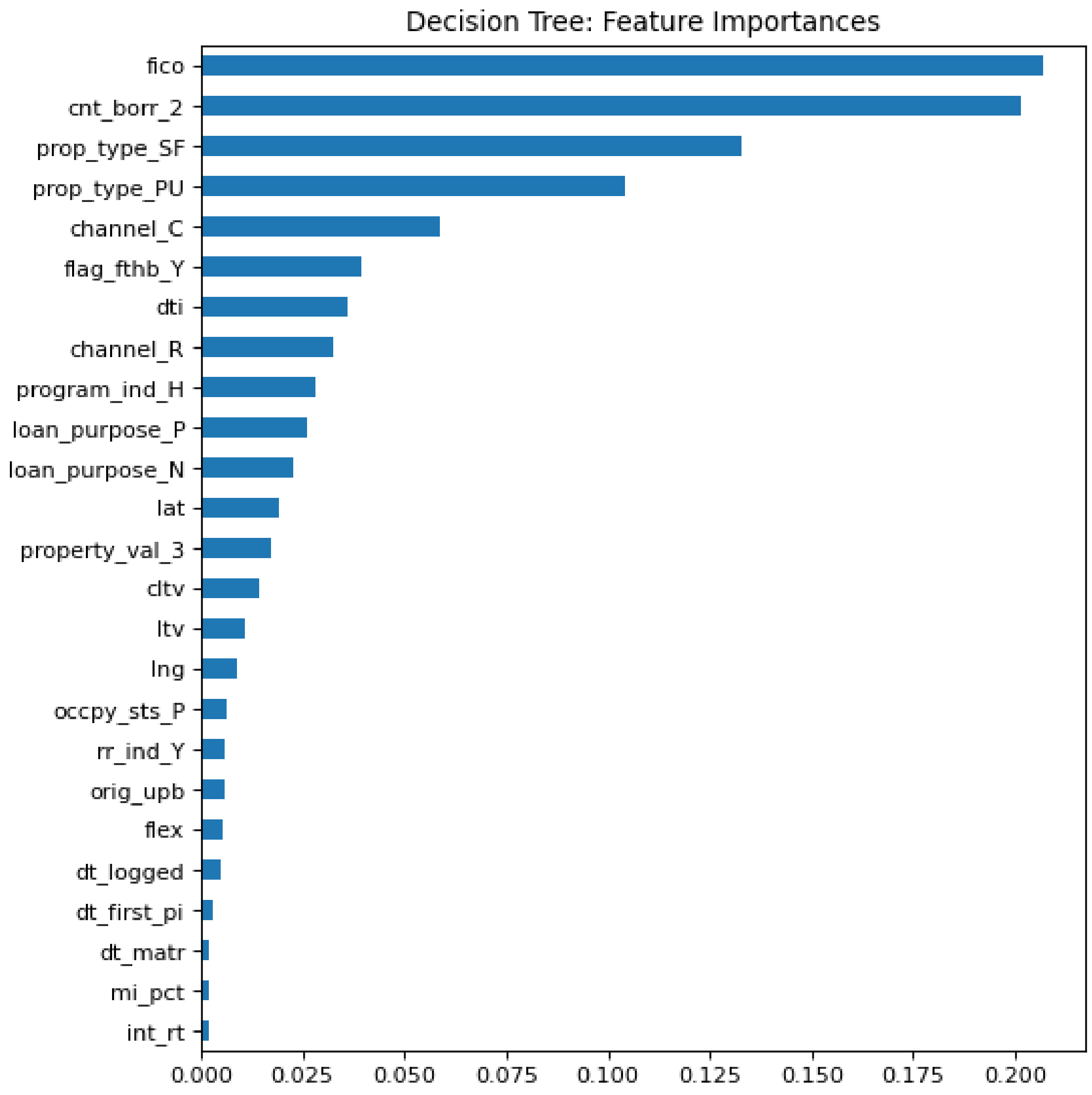
```
594/594                    1s 801us/step
```



The confusion matrix for the test set, which has not been used at all throughout this process, confirms our argument in terms of performance. We will proceed using the Neural Network model to ensure balance in the way that Freddie Mac priorities the identification false positives (incorrectly assigning an individual who does not default to the default category) and false negatives (incorrectly assigning an individual who defaults to the non-default category). However, there is still useful insight that can be extracted from the random forest model.

4.1 Feature Importance

[128]:
```
importances = pd.Series(
    rf_model.feature_importances_, index=feature_names
).sort_values(ascending=True)

nonzero_importances = importances[importances > 0.001]

# Plot the feature importances
fig, ax = plt.subplots(figsize=(7,7))
ax = nonzero_importances.plot.barh()
ax.set_title("Decision Tree: Feature Importances")
ax.figure.tight_layout()
```



As expected, `fico` is also one of the highest indicators of whether a loan will default. Then, `cnt_borr` is the second most important feature under this model, which supports EDA. When there is more than one borrower on the loan, there is less risk involved. The category `prop_type` follows this. This is another feature that was highlighted in the EDA as having a disproportionate number of defaulters. However, although Manufactured Homes were raised as important in the EDA, they are not viewed the same way under the model, perhaps due to the small number of observations.

Some of the features had an importance factor of zero. These include `orig_loan_term`, `cnt_units`, `mi_cancel_ind` and `flag_sc`.

4.1.1 Active Loans

```
[74]: active_df = active[[col for col in active.columns if col != 'loan_status'] +␣
      ↪['loan_status']]

      X_active, y_active = active_df.drop(['loan_status'], axis=1),␣
      ↪nonactive['loan_status']

      X_active_e = ct.transform(X_active.values)
      X_active_e_nn = min_max_scaler.transform(X_active_e)

      # predict active loans
      active_pred = nn_model_deep.predict(X_active_e_nn)
      active_plot_df = active.copy()
      active_plot_df["preds"] = 1-active_pred
```

```
2289/2289                  2s 1ms/step
```

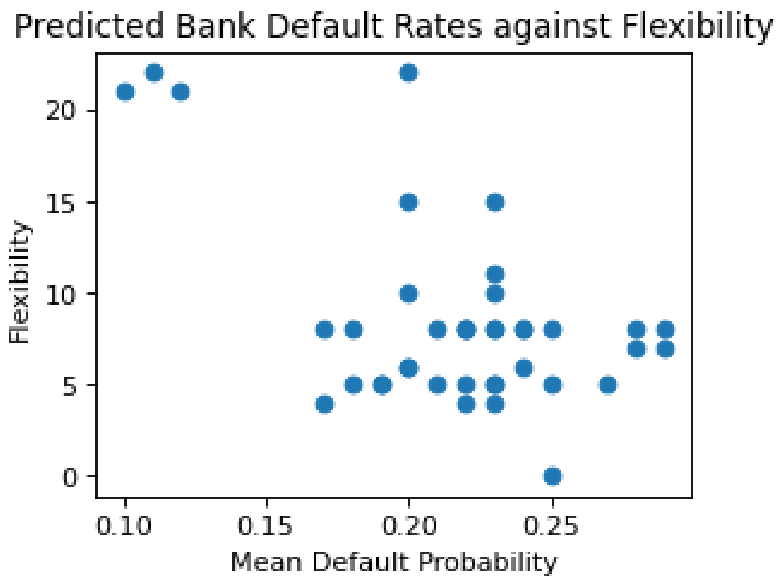
The data frame `active_plot_df` contains a prediction for each customer with an existing loan. This prediction is between 0 and 1, where 1 is represents a default and 0 represents a non-default. The company can use this to monitor current loans and perhaps determine where more attention could be paid to ensure that the loan is not defaulted. A threshold can be used if a decision needs to be made. Our final model uses a threshold of 0.5.

```
[134]: """ Finding Bank Scores """

       bank_scores = pd.DataFrame(index=active_plot_df['servicer_name'].unique()[1:],␣
       ↪columns=['Mean Default Prob.', 'Flexibility'])

       for bank in active_plot_df['servicer_name'].unique()[1:]:
           filtered = active_plot_df[active_plot_df['servicer_name'] == bank]
           avg_prob = round(filtered['preds'].mean(),2)
           flex = filtered.iloc[0]['flex']
           bank_scores.loc[bank, 'Mean Default Prob.'] = avg_prob
           bank_scores.loc[bank, 'Flexibility'] = round(flex)



       active_plot_df['flex'].value_counts()

       bank_scores = bank_scores.sort_values(by='Mean Default Prob.', ascending=False)

       display(bank_scores.iloc[:10])
       fig, ax = plt.subplots(figsize=(4,3))
       ax.scatter(bank_scores['Mean Default Prob.'], bank_scores['Flexibility'])
       ax.set_xlabel('Mean Default Probability')
       ax.set_ylabel('Flexibility')
       ax.set_title('Predicted Bank Default Rates against Flexibility')
```

```
                                       Mean Default Prob. Flexibility
```

```
SPECIALIZED LOAN SERVICING LLC                 0.29          8
AMERIHOME MORTGAGE COMPANY, LLC                0.29          7
CITIMORTGAGE, INC.                             0.28          7
FIFTH THIRD BANK, NATIONAL ASSOCIATION         0.28          8
AMERISAVE MORTGAGE CORPORATION                 0.27          5
LAKEVIEW LOAN SERVICING, LLC                   0.25          8
COLONIAL SAVINGS, F.A.                         0.25          5
NATIONSTAR MORTGAGE LLC DBA MR. COOPER         0.25          0
GUARANTEED RATE, INC.                          0.24          8
AURORA FINANCIAL GROUP, INC.                   0.24          6
```

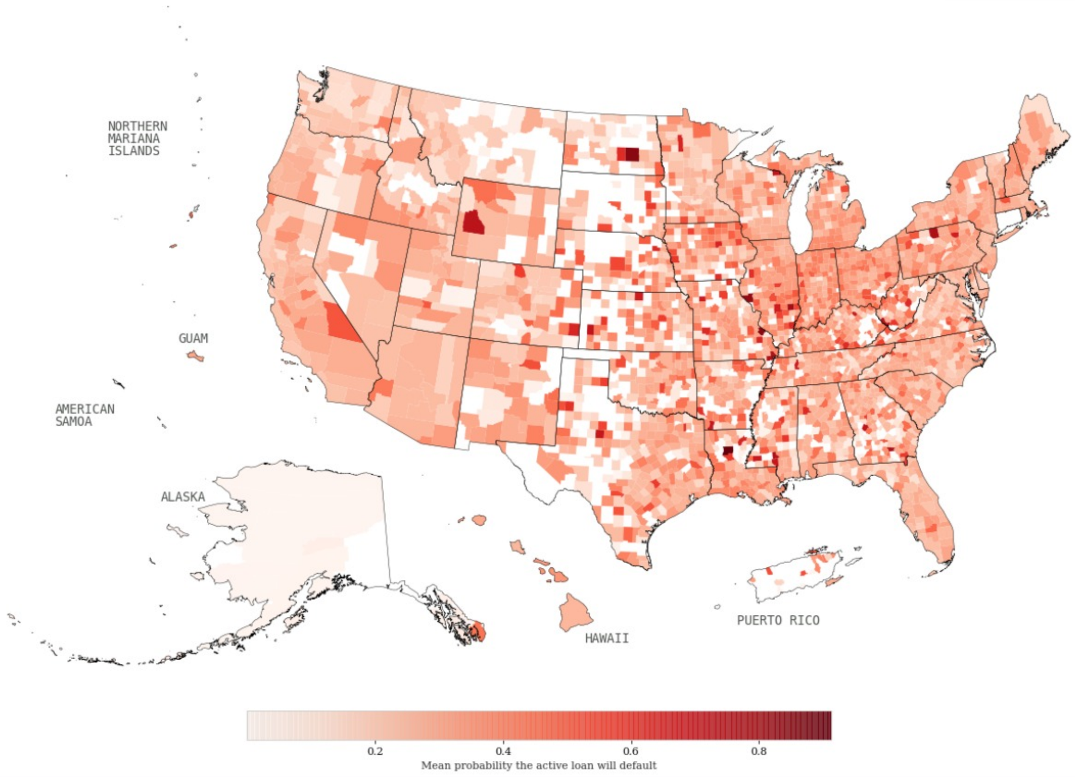
```
[134]: Text(0.5, 1.0, 'Predicted Bank Default Rates against Flexibility')
```



We have offered an average ranking for each bank, displaying the top 10 highest risk servicers, which the company might like to use, if reconsidering its service providers. The higher probability ratings indicate a higher risk of defaulting. When we look at the plot below, this demonstrates our motativation to add this feature engineering step of flexibility. Banks with more flexbility often have a lower mean default probability.

```
[ ]: Image('png/DEFAULT_PROB.jpg', width=900)
```

```
[ ]:
```



This plot provides an outline of mean probabilities of defaults across all areas. This has been provided so the company can idenfity areas of risk, which could help when considering loans.

5 Discussion & Conclusions

We have been presented with a large dataset of fixed interest rate loans with the task of identiftying which of the currently active loans might default in the future. We were given a wide range of 32 categorical and numerical features with which we could build a model to do this. We have utilised Synthetic Minority Over-sampling Technique (SMOTE) to fix the sample inequality whilst minimising the overfitting caused by traditional oversamping. The size and non-linearity of the dataset poses a problem for Logistic Regression and SVMs. After considering a range of non-linear models, we have chosen a Neural Network, which can be used to make predictions on existing loans and new loans to come. The main detail that influenced our decisions is balancing importance of correct classification. While prioritising a high default classification rate, we have also ensured that our model does not misclassify too many non-defaults, which will also cost the company. The Neural Network offers a computational advantage by using deep learning to determine a rating for each loan, identifying the risk of default. However, one drawback of this is that the results from this model are not as intuitive. This is why we propose that the company also make use of the Random Forest model to identify the factors associated to higher default rates. This information could be used to inform future company policy.

6 References

Papers:

- Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs) - by Djork-Arné Clevert and Thomas Unterthiner and Sepp Hochreiter - 2016 (*https://arxiv.org/abs/1511.07289*)
- Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift - by Sergey Ioffe and Christian Szegedy - 2015 (*https://arxiv.org/abs/1502.03167*)
- Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification - by Kaiming He and Xiangyu Zhang and Shaoqing Ren and Jian Sun - 2015 (https://arxiv.org/abs/1502.01852)

Code that was adapted in order to create maps:

https://dev.to/oscarleo/how-to-create-data-maps-of-the-united-states-with-matplotlib-p9i

Link to dataset used to provide latitude and longitude values for the zip codes:

https://simplemaps.com/data/us-zips#anchor_centroid

[12]:
```
# Run the following to render to PDF
!jupyter nbconvert --to pdf project2prev.ipynb
```

```
[NbConvertApp] Converting notebook project2prev.ipynb to pdf
^C
Traceback (most recent call last):
  File "/Users/alfie/anaconda3/lib/python3.12/subprocess.py", line 1209, in
communicate
    stdout, stderr = self._communicate(input, endtime, timeout)
                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

  File "/Users/alfie/anaconda3/lib/python3.12/subprocess.py", line 2113, in
_communicate
    ready = selector.select(timeout)
            ^^^^^^^^^^^^^^^^^^^^^^^^

  File "/Users/alfie/anaconda3/lib/python3.12/selectors.py", line 415, in select
    fd_event_list = self._selector.poll(timeout)
                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^

KeyboardInterrupt

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/Users/alfie/anaconda3/bin/jupyter-nbconvert", line 11, in <module>
    sys.exit(main())
             ^^^^^^

  File "/Users/alfie/anaconda3/lib/python3.12/site-
packages/jupyter_core/application.py", line 283, in launch_instance
    super().launch_instance(argv=argv, **kwargs)
  File "/Users/alfie/anaconda3/lib/python3.12/site-
packages/traitlets/config/application.py", line 1075, in launch_instance
```