Fannie Mae

In this code, we will be building a model to predict default probabilities for a set of loans that Fannie Mae holds. This data set is very large, and you may get a popup at times saying that the "Kernel Died". If you do, that likely means you are out of memory. You will be able to use a larger virtual machine to work with this data, but depending on how many variables you use and how much feature engineering you do, you may still run out of memory on the larger machine.

If you choose to work on this data with the larger virtual machine, please be very conscious about logging out after you finish (by going to "File -> Log Out" in the toolbar). That will allow the virtual machine to be shut down as soon as you are done with it, saving the university money.

Moons of Jupyter Model Overview

We have left the majority of the original code untouched. We changed any unused code to "raw" to speed up runtime when starting up the Kernal.

Below is the summary of changes and decisions we made:

Engineering features:

- Adding new variable "LOG_OV" to scale the coefficients
- Dropped 2005, 2006, 2007, and 2008 as we felt like these year had bad data due to fradunant loan origination practices. We thought about creating a dummy variable, but if the data is truly bad, we weren't sure how a dummy would control for this aspect

Imputing: we did not change the imputing methodology, but with further time and technical knowledge we would concentrate on the credit score metrics (borrower and co-borrower). We tried a few different things but simply couldn't get the code to work.

Variable selection: we experimented with adding and removing different variables. We were very conflicted with adding year and age of loan - if we had, our skill score jumped to almost 9%. However, given that we were selling a model used at the time of orgination, we thought that these variables would make the model unusable. We were not sure how creating dummy variables controled for this effect so we chose the simpler approach.

Model selection: we chose just to use boosted trees rather than ensembling multiple models together. While we lost predictive power from not ensembling, we felt like having a simple way to communicate feature importance to the client and their loan originators was extremely important. We used a grid search to tune the final model parameters.

Scoring: For the scoring, we tried using the skill score metric (in the import packages you can see we imported "make_scorer") but we couldn't make it work. Instead we put in the brier score directly ('neg_brier_score'), then confirmed that the skill score was rising as we made changes.

Revenue predictions: Because our pitch centered on cost savings from servicing fewer defaulted loans, we did not use the net loss prediction section. However, we still tried to create the net loss dataset as we knew we needed it to create the prediction output.

Outputting predictions: Because we dropped data, the "fresh" prediction dataset wouldn't match up with our predictions, and the prediction dataset wouldn't write to CSV.

Data used: After choosing all the model parameters, we ran and tuned the model on the large dataset (10%).

Import Packages

In [1]:

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import matplotlib as mpl
import seaborn as sns
import dill
import random

from sklearn.metrics import roc_auc_score, accuracy_score, precision_score, f1_scor
e, confusion_matrix, plot_roc_curve, make_scorer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder, FunctionTransformer, LabelEncoder,
StandardScaler, OrdinalEncoder
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import RandomForestClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split, GridSearchCV, KFold, cross_va
l_predict
from xgboost import XGBClassifier

from patsy import dmatrices, dmatrix, build_design_matrices

# Set number of CPU cores for parallel algorithms
import os
if "CPU_LIMIT" in os.environ:
    # If you are on JupyterHub, this gives you the right number of CPUs for your vi
rtual machine
    num_cpus = int(os.getenv("CPU_LIMIT").split('.')[0])
else:
    # If you are not on JupyterHub, this gives you the right number for your comput
er.
    num_cpus = os.cpu_count()
```

In [2]:

```
# This sets some nicer defaults for plotting.
# This must be run in a separate cell from importing matplotlib due to a bug.
params = {'legend.fontsize': 'large',
          'figure.figsize': (11.0, 11.0),
          'axes.labelsize': 'x-large',
          'axes.titlesize':'xx-large',
          'xtick.labelsize':'large',
          'ytick.labelsize':'large'}
mpl.rcParams.update(params)

# This makes it so that the pandas dataframes don't get truncated horizontally.
pd.options.display.max_columns = 200
```

Load the Data

Here we load the data. There are two pre-built data sets. The first is a 2% random sample of the full data set (split into training and testing), and it is about 397,000 rows in each of the training and testing set (or almost 800,000 rows total). The full data set is about 40,000,000 rows with 90% of the data being training and the other 10% for testing. To load the full data set, you should just set the variable `full_data_set` to `True` like this:

```
full_data_set = True
```

There is also a variable `p` that will allow you to choose how much of the training set to load in. You may find it both very difficult and potentially unnecessary to use the full training set to build your models. If so, you can select any proportion of the data set that you would like to use by setting `p` equal to the proportion. I.e., `p=.1` means use 10% of the full training set. If you try to use the full data set, be prepared for many common things (like `summarize_dataframe()`) to take quite a long time.

In the test set, you have the realizations as well (i.e. the dependent variable is in both the training and testing). This is because you will need to evaluate your model yourself given the size of the test set. The self-evaluated skill score will be our benchmark for the competition.

In [3]:

```
%%time

full_data_set = True

if not full_data_set:
    df_train = pd.read_csv('../Shared Data (Read Only)/Fannie Mae Data/FannieMaeSmallTrain.csv', sep='|')
    df_test = pd.read_csv('../Shared Data (Read Only)/Fannie Mae Data/FannieMaeSmallTest.csv', sep='|')

if full_data_set:
    # This p is the proportion of the training data you load.
    # You can set it anywhere from 0 to 1.
    p = .1
    random.seed(201)
    df_train = pd.read_csv('../Shared Data (Read Only)/Fannie Mae Data/FannieMaeTrain.csv',
                           sep='|',
                           skiprows=lambda i: i>0 and random.random() > p)
    df_test = pd.read_csv('../Shared Data (Read Only)/Fannie Mae Data/FannieMaeTest.csv', sep='|')
```

```
CPU times: user 47.1 s, sys: 2.74 s, total: 49.8 s
Wall time: 49.8 s
```

In [4]:

```
df_train.shape
```

Out[4]:

```
(3579919, 37)
```

In [5]:

```
df_test.shape
```

Out[5]:

```
(3977372, 37)
```

Summarize the Data

As we have done many times by now, we are going to summarize the data, see what missing values look like, and get a sense for what is a continuous variable and what is a categorical variable. As we have seen before, just because a variable is a number (i.e. `int64` for integers or `float64` for real numbers), it doesn't mean it's continuous. For example, below we see that `NUMBER_OF_BORROWERS` has 8 unique values. Those values are all numbers, so it is a `float64` variable, but that variable really is the number of borrowers on the loan, which is better thought of as a categorical variable. So, you should dig into the data glossary from the Fannie Mae website (https://www.fanniemae.com/portal/funding-the-market/data/loan-performance-data.html) (direct link to glossary is here (https://loanperformancedata.fanniemae.com/lppub-docs/FNMA_SF_Loan_Performance_Glossary.pdf)) in order to understand what these variables mean.

The data set contains all of the variables from the acquisition file, and we have pre-computed some additional independent variables (i.e. variables we can use to predict) that are not in the glossary:

- `CREDIT_SCORE_MIN` : The minimum of the borrower and the co-borrowers credit score.
- `ORIGINAL_VALUE` : The original value of the house. Computed from `ORIGINAL_LTV` (original loan to value) and `ORIGINAL_UPB` (original unpaid balance).
- `MSA_NAME` : The name of the MSA.
- `CENSUS_2010_POP` : The population in 2010 for the MSA.

We have also pulled in or pre-computed a number of variables that we cannot use for our prediction. This is because these variables are computed from the performance data while we are computing our dependent variable and if we used them, this would be future data leaking into the past (i.e. data we wouldn't have at decision time). These variables are:

- `ZERO_BALANCE_CODE` : The code indicated in the data set for when the mortgage is at zero balance.
- `LOAN_AGE` : The age of the loan in the data set.
- `NET_LOSS` : The net loss of a loan that has experienced a credit event (i.e. default).
- `NET_SEVERITY` : The net loss divided by the unpaid balance (i.e. percent loss).
- `LAST_STAT` : a letter indicating the last status of the loan in the data set. The values are as follows:
  - `P` : Prepaid or matured loan.
  - `T` : Third party sale of loan.
  - `S` : Short sale.
  - `R` : Repurchased loan.
  - `F` : Deed-in-lieu of foreclosure.
  - `N` : Note sale.
  - `L` : Reperforming loan sale.
  - `X` : Loan is delinquent, but status is unknown.
  - `9` : The loan has been delinquent for at least 9 months.
  - `1 - 8` : The loan has been delinquent for this many months.
  - `C` : The loan is current.
- `LOAN_MODIFICATION_COSTS` : Costs due to modifying loan terms (i.e. reducing interest rate or forgiving principle).
- `TOTAL_LOSSES` : `NET_LOSS` plus `LOAN_MODIFICATION_COSTS` .

Here, we are predicting the default rate, which we do not actually have. We will create this using our `LAST_STAT` variable. Default events are when `LAST_STAT` is one of `F`, `S`, `T`, or `N`. We will compute a new dummy variable called `DEFAULT_FLAG` which will be `1` when `LAST_STAT` is one of these values. We will drop all other variables that are computed using the performance data to avoid the possibility of accidentally using them to build our model.

In [ ]:

```
#Keeping in loan age to use to potentially use
```

In [6]:

```
if 'ZERO_BALANCE_CODE' in df_train:
    df_train['DEFAULT_FLAG'] = df_train['LAST_STAT'].isin(['F', 'S', 'T', 'N']).ast
ype(int)
    df_train.drop(['ZERO_BALANCE_CODE', 'NET_LOSS', 'NET_SEVERITY', 'LAST_STAT', 'L
OAN_MODIFICATION_COSTS', 'TOTAL_LOSSES'],
                  axis=1,
                  inplace=True)
if 'ZERO_BALANCE_CODE' in df_test:
    df_test['DEFAULT_FLAG'] = df_test['LAST_STAT'].isin(['F', 'S', 'T', 'N']).astyp
e(int)
    df_test.drop(['ZERO_BALANCE_CODE','NET_LOSS', 'NET_SEVERITY', 'LAST_STAT', 'LOA
N_MODIFICATION_COSTS', 'TOTAL_LOSSES'],
                  axis=1,
                  inplace=True)
```

In [7]:

```
def summarize_dataframe(df):
    """Summarize a dataframe, and report missing values."""
    missing_values = pd.DataFrame({'Variable Name': df.columns,
                                   'Data Type': df.dtypes,
                                   'Missing Values': df.isnull().sum(),
                                   'Unique Values': [df[name].nunique() for name in
df.columns]}
                                  ).set_index('Variable Name')
    with pd.option_context("display.max_rows", 1000):
        display(pd.concat([missing_values, df.describe(include='all').transpose()],
axis=1).fillna(""))
```

In [10]:

```
summarize_dataframe(df_train)
```

Variable Name	Data Type	Missing Values	Unique Values	count	unique	t
LOAN_IDENTIFIER	int64	0	3579919	3579919.0		
ORIGINATION_CHANNEL	object	0	3	3579919.0	3	
SELLER_NAME	object	0	95	3579919.0	95	OTH
ORIGINAL_INTEREST_RATE	float64	0	3479	3579919.0		
ORIGINAL_UPB	float64	0	921	3579919.0		
ORIGINAL_LOAN_TERM	int64	0	274	3579919.0		
ORIGINATION_DATE	object	0	234	3579919.0	234	20(07·
FIRST_PAYMENT_DATE	object	0	235	3579919.0	235	20(09·
ORIGINAL_LTV	float64	0	97	3579919.0		
ORIGINAL_COMBINED_LTV	float64	0	156	3579919.0		
NUMBER_OF_BORROWERS	float64	874	10	3579045.0		
ORIGINAL_DTI	float64	68574	64	3511345.0		
BORROWER_CREDIT_SCORE	float64	14122	439	3565797.0		
FIRST_TIME_HOME_BUYER	object	0	3	3579919.0	3	
LOAN_PURPOSE	object	0	4	3579919.0	4	
PROP_TYPE	object	0	5	3579919.0	5	
NUMBER_OF_UNITS	int64	0	4	3579919.0		
OCCUPANCY_TYPE	object	0	3	3579919.0	3	
STATE	object	0	54	3579919.0	54	
ZIP_CODE_SHORT	int64	0	960	3579919.0		
PRIMARY_MORTGAGE_INSURANCE_PERCENT	float64	2939154	42	640765.0		
PRODUCT_TYPE	object	0	1	3579919.0	1	FI
COBORROWER_CREDIT_SCORE	float64	1796520	414	1783399.0		
MORTGAGE_INSURANCE_TYPE	float64	2939154	3	640765.0		
RELOCATION_MORTGAGE	object	0	2	3579919.0	2	
CREDIT_SCORE_MIN	float64	8873	452	3571046.0		
ORIGINAL_VALUE	float64	0	31817	3579919.0		
MSA	int64	0	406	3579919.0		
LOAN_AGE	float64	0	242	3579919.0		

Variable Name	Data Type	Missing Values	Unique Values	count	unique	t
MSA_NAME	object	452339	381	3127580.0	381	N Yo Newa Jers C NY-N
CENSUS_2010_POP	float64	452339	381	3127580.0		
DEFAULT_FLAG	int64	0	2	3579919.0		

In [11]:

```
summarize_dataframe(df_test)
```

Variable Name	Data Type	Missing Values	Unique Values	count	unique	t
LOAN_IDENTIFIER	int64	0	3977372	3977372.0		
ORIGINATION_CHANNEL	object	0	3	3977372.0	3	
SELLER_NAME	object	0	95	3977372.0	95	OTH
ORIGINAL_INTEREST_RATE	float64	0	3539	3977372.0		
ORIGINAL_UPB	float64	0	922	3977372.0		
ORIGINAL_LOAN_TERM	int64	0	283	3977372.0		
ORIGINATION_DATE	object	0	234	3977372.0	234	20(07·
FIRST_PAYMENT_DATE	object	0	235	3977372.0	235	20(09·
ORIGINAL_LTV	float64	0	97	3977372.0		
ORIGINAL_COMBINED_LTV	float64	0	159	3977372.0		
NUMBER_OF_BORROWERS	float64	970	10	3976402.0		
ORIGINAL_DTI	float64	76154	64	3901218.0		
BORROWER_CREDIT_SCORE	float64	15859	448	3961513.0		
FIRST_TIME_HOME_BUYER	object	0	3	3977372.0	3	
LOAN_PURPOSE	object	0	4	3977372.0	4	
PROP_TYPE	object	0	5	3977372.0	5	
NUMBER_OF_UNITS	int64	0	4	3977372.0		
OCCUPANCY_TYPE	object	0	3	3977372.0	3	
STATE	object	0	54	3977372.0	54	
ZIP_CODE_SHORT	int64	0	968	3977372.0		
PRIMARY_MORTGAGE_INSURANCE_PERCENT	float64	3266569	41	710803.0		
PRODUCT_TYPE	object	0	1	3977372.0	1	FI
COBORROWER_CREDIT_SCORE	float64	1994390	416	1982982.0		
MORTGAGE_INSURANCE_TYPE	float64	3266569	3	710803.0		
RELOCATION_MORTGAGE	object	0	2	3977372.0	2	
CREDIT_SCORE_MIN	float64	9760	460	3967612.0		
ORIGINAL_VALUE	float64	0	32263	3977372.0		
MSA	int64	0	406	3977372.0		
LOAN_AGE	float64	0	243	3977372.0		

Variable Name	Data Type	Missing Values	Unique Values	count	unique	t
MSA_NAME	object	502361	381	3475011.0	381	N Yo Newa Jers C NY-N
CENSUS_2010_POP	float64	502361	381	3475011.0		
DEFAULT_FLAG	int64	0	2	3977372.0		

Notice that the training and test sets are practically identical (in fact they are practically the same size). Moreover, you have the realizations for each of the test data points. Given the size of the dataset, it is infeasible to submit predictions, so you will evaluate your performance directly on the test set. You should be wary of using the test set too much or you might overfit the test set.

Engineer Row Based Features

As long as we don't need to use any information other than what we have, we can go ahead and engineer new features on the full training set. Here we will convert the `ORIGINATION_DATE` column from an object (i.e., string) to a formal datetime object in Python.

In [12]:

```
df_train['ORIGINATION_DATE'] = pd.to_datetime(df_train['ORIGINATION_DATE'], format=
'%Y-%m-%d')

df_test['ORIGINATION_DATE'] = pd.to_datetime(df_test['ORIGINATION_DATE'], format='%
Y-%m-%d')
```

We will then use `ORIGINATION_DATE` to compute the year in which the loan was originated.

In [13]:

```
df_train['YEAR'] = df_train['ORIGINATION_DATE'].dt.year
df_test['YEAR'] = df_test['ORIGINATION_DATE'].dt.year
```

If you have any other features that you'd like to engineer that can be done with a single row, you can do that here as well.

In [14]:

```
#Additions or changes to features
```

In [15]:

```
#Scaling original value variable to remove extremely large coefficients
df_train['LOG_OV']=np.log(df_train['ORIGINAL_VALUE'])
df_test['LOG_OV']=np.log(df_test['ORIGINAL_VALUE'])
```

In [16]:

```
#Dropping years of biased data where there were poor origination practices or fraud
#Dropped from 298302 to 258744, 39558 total in the test set

df_train.drop(df_train.loc[df_train['YEAR']==2005].index, inplace=True)
df_test.drop(df_test.loc[df_test['YEAR']==2005].index, inplace=True)

df_train.drop(df_train.loc[df_train['YEAR']==2006].index, inplace=True)
df_test.drop(df_test.loc[df_test['YEAR']==2006].index, inplace=True)

df_train.drop(df_train.loc[df_train['YEAR']==2007].index, inplace=True)
df_test.drop(df_test.loc[df_test['YEAR']==2007].index, inplace=True)

df_train.drop(df_train.loc[df_train['YEAR']==2008].index, inplace=True)
df_test.drop(df_test.loc[df_test['YEAR']==2008].index, inplace=True)
```

Split Into Training and Validation

Now we will split the train dataset into a smaller training and a validation set, as is best practice.

In [17]:

```
df_smaller_train, df_validation = train_test_split(df_train, test_size = 0.25, rand
om_state = 201)
```

There is a bug that gives a warning later on in the code which you can fix by making copies of the above dataframe. This step is unnecessary, but it avoids showing a warning, so I'm going to include it.

In [18]:

```
df_smaller_train = df_smaller_train.copy()
df_validation = df_validation.copy()
```

Impute Missing Values

Now, we will impute missing values. We will define the same `CategoricalImputer` from CarvanaStarter-II.

In [19]:

```
from sklearn.base import BaseEstimator, TransformerMixin

class CategoricalImputer(BaseEstimator, TransformerMixin):
    """
    Custom defined imputer for categorical data. This allows you to specify an
    other class where any category that doesn't meet the requirements necessary to
    be in
    """

    def __init__(self, other_threshold=0,
                 other_label="OTHER",
                 missing_first=True,
                 missing_values=np.nan,
                 strategy='constant',
                 fill_value="MISSING",
                 verbose=0,
                 copy=True,
                 add_indicator=False):
        self._other_threshold = other_threshold
        self._other_label = other_label
        self._missing_first = missing_first
        if hasattr(missing_values, "__iter__"):
            self._missing_values = missing_values
        else:
            self._missing_values = [missing_values]
        self._imputer = SimpleImputer(missing_values=missing_values, strategy=strat
egy, fill_value=fill_value, verbose=verbose, copy=copy, add_indicator=add_indicator
)
        self._column_categories = {}

    def fit(self, X, y=None):
        if type(self._other_threshold) == int or type(self._other_threshold) == flo
at:
            other_threshold = [self._other_threshold]*len(X.columns)
        elif len(self._other_threshold) == len(X.columns):
            other_threshold = self._other_threshold
        else:
            raise TypeError("other_threshold must be either a single number or a li
st of numbers equal to the number of columns.")

        i = 0
        X = X.copy()
        X = X[:].astype(object)
        if self._missing_first:
            X = pd.DataFrame(self._imputer.fit_transform(X), columns=X.columns, ind
ex=X.index)
        column_categories = {}
        for column in X.columns:
            if other_threshold[i] < 1:
                other_threshold[i] = other_threshold[i]*X[column].shape[0]

            value_counts = X[column].value_counts()
            categories = [category for category in value_counts.index if value_coun
ts.loc[category] >= other_threshold[i]]
            if value_counts.iloc[-1] >= other_threshold[i]:
```

```
                categories[-1] = self._other_label
            else:
                categories.append(self._other_label)

            self._column_categories[column] = categories
            i = i + 1

        return self

    def transform(self, X, y=None):
        X = X.copy()
        X = X[:].astype(object)
        if self._missing_first:
            X = pd.DataFrame(self._imputer.fit_transform(X), columns=X.columns, ind
ex=X.index)
        for column in X.columns:
            X.loc[~X[column].isin(self._column_categories[column]) & ~X[column].isi
n(self._missing_values), column] = self._other_label
        return pd.DataFrame(self._imputer.fit_transform(X), columns=X.columns, inde
x=X.index)[:].astype(str)
```

It is often helpful to list all of the variables, and then create new lists that break up variables into things you want to impute together.

In [20]:

```
list(df_smaller_train.columns)
```

Out[20]:

```
['LOAN_IDENTIFIER',
 'ORIGINATION_CHANNEL',
 'SELLER_NAME',
 'ORIGINAL_INTEREST_RATE',
 'ORIGINAL_UPB',
 'ORIGINAL_LOAN_TERM',
 'ORIGINATION_DATE',
 'FIRST_PAYMENT_DATE',
 'ORIGINAL_LTV',
 'ORIGINAL_COMBINED_LTV',
 'NUMBER_OF_BORROWERS',
 'ORIGINAL_DTI',
 'BORROWER_CREDIT_SCORE',
 'FIRST_TIME_HOME_BUYER',
 'LOAN_PURPOSE',
 'PROP_TYPE',
 'NUMBER_OF_UNITS',
 'OCCUPANCY_TYPE',
 'STATE',
 'ZIP_CODE_SHORT',
 'PRIMARY_MORTGAGE_INSURANCE_PERCENT',
 'PRODUCT_TYPE',
 'COBORROWER_CREDIT_SCORE',
 'MORTGAGE_INSURANCE_TYPE',
 'RELOCATION_MORTGAGE',
 'CREDIT_SCORE_MIN',
 'ORIGINAL_VALUE',
 'MSA',
 'LOAN_AGE',
 'MSA_NAME',
 'CENSUS_2010_POP',
 'DEFAULT_FLAG',
 'YEAR',
 'LOG_OV']
```

Here, we really need to be careful when we are imputing values. A missing value very well might be meaningful. For example, one of the columns that has missing values is `BORROWER_CREDIT_SCORE` . Take a look at the data glossary here (https://loanperformancedata.fanniemae.com/lppub-docs/FNMA_SF_Loan_Performance_Glossary.pdf) to see what that means. What would you impute `BORROWER_CREDIT_SCORE` as?

We will create a few different imputers to use.

In [21]:

```
imputer_mean = SimpleImputer(missing_values=np.nan, strategy='mean')
imputer_zero = SimpleImputer(missing_values=np.nan, strategy='constant', fill_value
=0)
categorical_imputer = CategoricalImputer(other_threshold=.01)
```

We have an imputer that replaces missing values with the mean, one that replaces missing values with `0`, and one that imputes for categorical variables that lumps in categories that is less than 1% of the data in the `OTHER` category.

Let's make a list of which variables we would like to use on these.

In [22]:

```
continuous_mean = ['ORIGINAL_INTEREST_RATE',
                   'ORIGINAL_UPB',
                   'ORIGINAL_LOAN_TERM',
                   'ORIGINAL_LTV',
                   'ORIGINAL_COMBINED_LTV',
                   'ORIGINAL_DTI',
                   'ORIGINAL_VALUE',
                   'YEAR']

continuous_zero = ['PRIMARY_MORTGAGE_INSURANCE_PERCENT',
                   'BORROWER_CREDIT_SCORE',
                   'COBORROWER_CREDIT_SCORE',
                   'CREDIT_SCORE_MIN']

continuous_variables = continuous_mean + continuous_zero

categorical_variables = ['ORIGINATION_CHANNEL',
                         'SELLER_NAME',
                         'NUMBER_OF_BORROWERS',
                         'FIRST_TIME_HOME_BUYER',
                         'LOAN_PURPOSE',
                         'PROP_TYPE',
                         'NUMBER_OF_UNITS',
                         'OCCUPANCY_TYPE',
                         'STATE',
                         'ZIP_CODE_SHORT',
                         'PRODUCT_TYPE',
                         'MORTGAGE_INSURANCE_TYPE',
                         'RELOCATION_MORTGAGE',
                         'MSA',
                         'MSA_NAME',
                         'CENSUS_2010_POP']
```

Note that I list nearly all variables above between the three categories, even though many of the variables do not have missing values. This is because any variable that we want to include in the model will always have to have a complete set of data whenever we predict, and we never know when a new row in the data will come in that has a missing value you have never seen before. If we go ahead and fit imputers for all of the columns, then we can safely impute on this new row without missing a beat.

However, I do not have a few variables in there. First, I did not include `DEFAULT_FLAG`. This is the variable we are trying to predict, and we should never impute it. I also did not include `LOAN_IDENTIFIER`. This is because this is a unique value that identifies the loans, and if it is missing, then there is probably something wrong with that row of your data, and you should probably get an error if you try to predict on it.

I also leave out `ORIGINATION_DATE` because that is a `datetime` object and none of our imputers can handle it. Be careful using these as features in a model. Though if you create features based on them, you can likely impute on those just fine.

I chose to impute a zero to `PRIMARY_MORTGAGE_INSURANCE_PERCENT`, `BORROWER_CREDIT_SCORE`, `COBORROWER_CREDIT_SCORE`, and `CREDIT_SCORE_MIN`. Take a look at the data glossary to figure out why. Feel free to make a different choice if you think it would be better.

Now that we have identified our categories of variables, we can fit the imputers and transform the data.

In [23]:

```
imputer_mean.fit(df_smaller_train[continuous_mean])
df_smaller_train[continuous_mean] = imputer_mean.transform(df_smaller_train[continuous_mean])
df_validation[continuous_mean] = imputer_mean.transform(df_validation[continuous_mean])
```

In [24]:

```
imputer_zero.fit(df_smaller_train[continuous_zero])
df_smaller_train[continuous_zero] = imputer_zero.transform(df_smaller_train[continuous_zero])
df_validation[continuous_zero] = imputer_zero.transform(df_validation[continuous_zero])
```

In [25]:

```
categorical_imputer.fit(df_smaller_train[categorical_variables])
df_smaller_train[categorical_variables] = categorical_imputer.transform(df_smaller_train[categorical_variables])
df_validation[categorical_variables] = categorical_imputer.transform(df_validation[categorical_variables])
```

Now that we have imputed everything, we can check to see that we no longer have missing values.

In [26]:

```
summarize_dataframe(df_smaller_train)
```

```
/opt/conda/lib/python3.7/site-packages/ipykernel_launcher.py:9: FutureW
arning: Treating datetime data as categorical rather than numeric in `.
describe` is deprecated and will be removed in a future version of pand
as. Specify `datetime_is_numeric=True` to silence this warning and adop
t the future behavior now.
  if __name__ == '__main__':
```

Variable Name	Data Type	Missing Values	Unique Values	count	unique
LOAN_IDENTIFIER	int64	0	2329420	2329420.0	
ORIGINATION_CHANNEL	object	0	3	2329420.0	3
SELLER_NAME	object	0	13	2329420.0	13
ORIGINAL_INTEREST_RATE	float64	0	3112	2329420.0	
ORIGINAL_UPB	float64	0	897	2329420.0	
ORIGINAL_LOAN_TERM	float64	0	266	2329420.0	
ORIGINATION_DATE	datetime64[ns]	0	186	2329420.0	186
FIRST_PAYMENT_DATE	object	0	189	2329420.0	189
ORIGINAL_LTV	float64	0	97	2329420.0	
ORIGINAL_COMBINED_LTV	float64	0	153	2329420.0	
NUMBER_OF_BORROWERS	object	0	3	2329420.0	3
ORIGINAL_DTI	float64	0	65	2329420.0	
BORROWER_CREDIT_SCORE	float64	0	426	2329420.0	
FIRST_TIME_HOME_BUYER	object	0	3	2329420.0	3
LOAN_PURPOSE	object	0	4	2329420.0	4
PROP_TYPE	object	0	4	2329420.0	4
NUMBER_OF_UNITS	object	0	3	2329420.0	3
OCCUPANCY_TYPE	object	0	3	2329420.0	3
STATE	object	0	31	2329420.0	31
ZIP_CODE_SHORT	object	0	2	2329420.0	2
PRIMARY_MORTGAGE_INSURANCE_PERCENT	float64	0	43	2329420.0	
PRODUCT_TYPE	object	0	1	2329420.0	1
COBORROWER_CREDIT_SCORE	float64	0	397	2329420.0	
MORTGAGE_INSURANCE_TYPE	object	0	4	2329420.0	4
RELOCATION_MORTGAGE	object	0	2	2329420.0	2
CREDIT_SCORE_MIN	float64	0	433	2329420.0	
ORIGINAL_VALUE	float64	0	30225	2329420.0	
MSA	object	0	24	2329420.0	24
LOAN_AGE	float64	0	242	2329420.0	
MSA_NAME	object	0	23	2329420.0	23
CENSUS_2010_POP	object	0	23	2329420.0	23

Variable Name	Data Type	Missing Values	Unique Values	count	unique
DEFAULT_FLAG	int64	0	2	2329420.0	
YEAR	float64	0	16	2329420.0	
LOG_OV	float64	0	29422	2329420.0	

Set up the scoring rule

The brier score is a common "scoring rule" for binary predictions. It is really just 1 minus the mean squared error of the prediction (where the prediction is a probability). We create a `brier_score` function below. A 1 would be perfectly predicting every default, while a 0 would be getting every prediction completely wrong.

In [27]:

```
from sklearn.metrics import brier_score_loss
def brier_score(realizations, predictions):
    this_brier_score = 1 - brier_score_loss(realizations, predictions)
    return this_brier_score
```

We always want to compare against some kind of baseline. A natural baseline here would be the "naive" forecast of just predicting that every loan has a a default rate equal to the average default rate in the training data. The below function predicts your "skill score", or the percent you do better than the naive prediction. A higher score is better, and a skill score of .08 would suggest that you did 8% better than the naive forecast.

This will be the metric we use for evaluating our performance.

In [28]:

```
def skill_score(realizations, predictions):
    naive = np.repeat(np.mean(df_train['DEFAULT_FLAG']), len(realizations))
    this_skill_score = (brier_score(realizations, predictions) - brier_score(realiz
ations, naive)) / (1 - brier_score(realizations, naive))
    return this_skill_score
```

Logistic regression

As we have seen, linear models like logistic regression typically require different kinds of features to be engineered from the variables. For example, we have seen that dummy variables are necessary in linear models, but not strictly necessary in tree based models (like classification trees and random forests). Additionally, interaction variables are the only way to get at certain effects in linear models, but tree based models can learn them directly. Another difference is with linear models you need to be more careful about is over-specifying the training data, i.e. if you have dummy variables you need to drop one column or the model is overspecified.

Therefore, it will be useful to do feature engineering separately for the two classes of models. Luckily `dmatrices` is a function that is very good at setting up data for a linear model. While we've used this a number of times before, this (https://patsy.readthedocs.io/en/latest/formulas.html#) link about how formulas work may be very helpful in constructing more complicated features including interactions and transformations.

We will start by specifying the formula we are going to use to generate our data. Again, refer to the link above for a refresher on how formulas work.

formula_logit = "DEFAULT_FLAG ~ standardize(ORIGINAL_LTV) + standardize(ORIGINAL_DTI) + standardize(BORROWER_CREDIT_SCORE) + FIRST_TIME_HOME_BUYER + STATE + standardize(YEAR) + FIRST_TIME_HOME_BUYER*standardize(BORROWER_CREDIT_SCORE)"

We by default have added four variables, and an interaction term. The five variables are `ORIGINAL_LTV`, `ORIGINAL_DTI`, `BORROWER_CREDIT_SCORE`, `STATE`, `YEAR`, and `FIRST_TIME_HOME_BUYER`. Then we interact `FIRST_TIME_HOME_BUYER` and `BORROWER_CREDIT_SCORE` with `FIRST_TIME_HOME_BUYER*BORROWER_CREDIT_SCORE`. Note that we use the `standardize()` function for any continuous variables. Standardizing continuous variables can significantly improve the performance of a regularized regression model, and it's generally recommended. There is no need to standardize dummy variables, and all of the other variables currently in the formula are dummy variables.

Another useful aspect of standardizing variables is in interpreting coefficients. If we don't standardize, then variables with larger variance (like `BORROWER_CREDIT_SCORE`) will necessarily have smaller coefficients because the number that multiplies the coefficients is just larger. Once all variables have been rescaled to have mean 0 and standard deviation of 1, we can say that the effect of one unit deviation from average is just the coefficient, which makes all the coefficients comparable.

Let's check to make sure our formula looks right (remember you have to get variable names exactly right, case and spelling are important!).

formula_logit

Now we can build our X and y to train our model.

y_logit_train, X_logit_train = dmatrices(formula_logit, df_smaller_train, return_type="dataframe")

Let's look at what it made.

y_logit_train

The `y_logit_train` is just whether or not each loan defaulted. Let's look at `X_logit_train`.

X_logit_train

What we see is that it created an intercept column, it created dummy variables for `FIRST_TIME_HOME_BUYER` (though there are only two columns, when there are three possible values, that is because it automatically drops one column), it created 29 columns dummy variables for `STATE`, it adds columns for our other continuous variables, and it adds two columns interacting the dummy columns of `FIRST_TIME_HOME_BUYER` with `BORROWER_CREDIT_SCORE`. Note that it automatically created dummy columns because `FIRST_TIME_HOME_BUYER` is an `object` type of data (which it automatically classifies as categorical variables).

Notice that it treats `YEAR` as a continuous variable, which might be what we want, but we also might want to treat `YEAR` as a dummy variable. `YEAR` is a `float64` in our data set, so it automatically assumes it is a continuous variable. However, you can force it to treat it as a dummy by replacing `YEAR` in the formula with `C(YEAR)`. The `C()` forces the variable to be treated as a dummy no matter what type it is.

Now we train our model. When you use the full data, this may take some time.

%%time rlr_model = LogisticRegression(C=1, random_state=201) rlr_train = rlr_model.fit(X_logit_train, y_logit_train)

You may have received a warning, but you can ignore it.

We can look at the size of the coefficients that it returned. Note that it is the absolute value of the coefficient that is most important. We can truly compare the size of the coefficients since the data has been scaled.

coef_df = pd.DataFrame({'Importance': rlr_train.coef_[0]}, index=X_logit_train.columns)
coef_df.reindex(coef_df.Importance.abs().sort_values(ascending=False).index)

Now we need to test on our validation set, but first we have to perform the *exact same transformations on it*. First, we will use our `build_design_matrices` function to get our `X_logit_validation` matrix. Remember the `y` values for the validation set are just `df_validation['DEFAULT_FLAG']`.

X_validation = build_design_matrices([X_logit_train.design_info], df_validation, return_type="dataframe")[0]

Now we can predict on the validation set.

rlr_pred = rlr_train.predict_proba(X_validation)[:,1]skill_score(df_validation['DEFAULT_FLAG'], rlr_pred)

Not great yet, but you can continue adding features and tuning using grid search.

Let's look at the ROC curve:

plot_roc_curve(rlr_model, X_validation, df_validation['DEFAULT_FLAG'])

Feature Engineering for Tree Based Models

In many ways, feature engineering is less important for tree based models, but that does not mean that it is entirely irrelevant. First, we can let our trees do less work finding the most important relationships if we can construct some of them for it, and we can save some splitting that the tree would normally have to do if we provide some dummy variables. We can also help point it in the right direction if we provide some interactions. First let's make a list of what to include broken up into three categories: continuous variables, ordinal encoded categorical variables, and dummy encoded categorical variables.

#Changed variables here Added: Log_OV (needed to scale down the original value, the value of the home impacts) Original interest rate (peak around 6.5%, another around 9%) Number of borrowers (1, 2, and 4 seem to matter, setting as ordinal because goes 1-10) Co-borrower credit score (not as important as it's not always present) Mortage insurance type (Property type (Occupancy type (investor properties fail more) Original combined LTV Loan age? Dropped: Original LTV (swapped with combined LTV because of the different combinations that was most significant) Did NOT add year to the model as year is not helpful for using the model to predict default rate of new loans. If we were using the model to buy exisiting loans, then year should be used. Did NOT add loan age for similar reasons as year, though loans typically start defaulting around age ~6 years, so more recent data is biased to not be defaulting.

In [29]:

```
continuous_features_trees = ['ORIGINAL_DTI', 'BORROWER_CREDIT_SCORE', 'LOG_OV','ORI
GINAL_INTEREST_RATE', 'COBORROWER_CREDIT_SCORE','ORIGINAL_COMBINED_LTV']
cat_ordinal_features_trees = ['STATE','NUMBER_OF_BORROWERS']
cat_dummy_features_trees = ['FIRST_TIME_HOME_BUYER','MORTGAGE_INSURANCE_TYPE','PROP
_TYPE','OCCUPANCY_TYPE']
```

original: continuous_features_trees = ['ORIGINAL_LTV', 'ORIGINAL_DTI', 'BORROWER_CREDIT_SCORE'] cat_ordinal_features_trees = ['STATE'] cat_dummy_features_trees = ['FIRST_TIME_HOME_BUYER']

Above we chose to create dummy variables for `FIRST_TIME_HOME_BUYER` because there are only three possible values, but we are ordinal encoding `STATE` because it would add 29 columns to our data if we tried to dummy encode it. That would throw off the random feature selection for things like random forests.

We can use these selected variables to make our training `X` matrix.

In [30]:

```
X_tree_train = df_smaller_train[continuous_features_trees + cat_ordinal_features_tr
ees]
y_tree_train = df_smaller_train['DEFAULT_FLAG']
```

That gives us the continuous features and the ordinal features, but we don't yet have the dummy features. The best way to get those is using the `dmatrix` command, which is really similar to `dmatrices`, but it doesn't give us back a `y` (which we already have from above. The syntax is very similar but we only specify the right hand side of the formula. For example, here we would do this.

In [31]:

```
formula_tree = "0 + " + " + ".join(cat_dummy_features_trees)
```

In [32]:

```
formula_tree
```

Out[32]:

```
'0 + FIRST_TIME_HOME_BUYER + MORTGAGE_INSURANCE_TYPE + PROP_TYPE + OCCU
PANCY_TYPE'
```

The `0` above says don't add in an intercept. We can also easily add in interaction variables if we want to.

In [33]:

```
formula_tree = "0 + " + " + ".join(cat_dummy_features_trees) + " + FIRST_TIME_HOME_
BUYER:BORROWER_CREDIT_SCORE"
```

In [34]:

```
formula_tree
```

Out[34]:

```
'0 + FIRST_TIME_HOME_BUYER + MORTGAGE_INSURANCE_TYPE + PROP_TYPE + OCCU
PANCY_TYPE + FIRST_TIME_HOME_BUYER:BORROWER_CREDIT_SCORE'
```

Note that I used `:` when adding the interaction term `FIRST_TIME_HOME_BUYER:BORROWER_CREDIT_SCORE`. This tells `dmatrix` to just add the interaction, not the individual variables (which we already have in our model).

Now we can create our dummy variables and interactions easily.

In [35]:

```
X_tree_train_patsy = dmatrix(formula_tree, df_smaller_train, return_type="datafram
e")
```

In [36]:

```
X_tree_train_patsy
```

Out[36]:

	FIRST_TIME_HOME_BUYER[N]	FIRST_TIME_HOME_BUYER[OTHER]	FIRST_TIME_HOME_BU
673766	0.0	0.0	
247368	1.0	0.0	
360192	0.0	0.0	
1521441	1.0	0.0	
1688133	1.0	0.0	
...	...	...	
601441	1.0	0.0	
3504099	1.0	0.0	
1469778	0.0	0.0	
1191496	0.0	0.0	
3085605	1.0	0.0	

2329420 rows × 14 columns

It did exactly what we were expecting, but we need to add it to our `X_tree_train` to get everything in the same place. We can concatenate the two dataframes with `pd.concat()`.

In [37]:

```
X_tree_train = pd.concat([X_tree_train, X_tree_train_patsy], axis=1)
```

This added the two dataframes together, and we can check to make sure it is what we want.

In [38]:

```
X_tree_train
```

Out[38]:

	ORIGINAL_DTI	BORROWER_CREDIT_SCORE	LOG_OV	ORIGINAL_INTEREST_RATE	CO
673766	29.000000	725.0	10.741817	4.125	
247368	32.679553	597.0	11.375724	7.625	
360192	33.000000	798.0	12.644328	3.500	
1521441	26.000000	736.0	12.362007	5.875	
1688133	31.000000	707.0	12.463541	6.125	
...	...	...	...	...	
601441	12.000000	798.0	12.355818	4.375	
3504099	29.000000	760.0	12.903674	6.750	
1469778	33.000000	743.0	12.211493	4.250	
1191496	30.000000	791.0	12.865318	4.125	
3085605	23.000000	708.0	12.546999	4.375	

2329420 rows × 22 columns

The final step is we need to ordinal encode any variables that are not dummies with `OrdinalEncoder`.

In [39]:

```
ordinal_encoder = OrdinalEncoder()
ordinal_encoder.fit(X_tree_train[cat_ordinal_features_trees])
X_tree_train[cat_ordinal_features_trees] = ordinal_encoder.transform(X_tree_train[c
at_ordinal_features_trees])
```

In [40]:

```
X_tree_train
```

Out[40]:

	ORIGINAL_DTI	BORROWER_CREDIT_SCORE	LOG_OV	ORIGINAL_INTEREST_RATE	CO
673766	29.000000	725.0	10.741817	4.125	
247368	32.679553	597.0	11.375724	7.625	
360192	33.000000	798.0	12.644328	3.500	
1521441	26.000000	736.0	12.362007	5.875	
1688133	31.000000	707.0	12.463541	6.125	
...	...	...	...	...	
601441	12.000000	798.0	12.355818	4.375	
3504099	29.000000	760.0	12.903674	6.750	
1469778	33.000000	743.0	12.211493	4.250	
1191496	30.000000	791.0	12.865318	4.125	
3085605	23.000000	708.0	12.546999	4.375	

2329420 rows × 22 columns

One final detail, xgboost does not like `[` or `]` in column names (e.g. `FIRST_TIME_HOME_BUYER_INDICATOR[N]`), so we will replace those with `(` and `)`.

In [41]:

```
X_tree_train.columns = X_tree_train.columns.str.replace('[', '(').str.replace(']',
')')
```

In [42]:

```
X_tree_train.columns
```

Out[42]:

```
Index(['ORIGINAL_DTI', 'BORROWER_CREDIT_SCORE', 'LOG_OV',
       'ORIGINAL_INTEREST_RATE', 'COBORROWER_CREDIT_SCORE',
       'ORIGINAL_COMBINED_LTV', 'STATE', 'NUMBER_OF_BORROWERS',
       'FIRST_TIME_HOME_BUYER(N)', 'FIRST_TIME_HOME_BUYER(OTHER)',
       'FIRST_TIME_HOME_BUYER(Y)', 'MORTGAGE_INSURANCE_TYPE(T.2.0)',
       'MORTGAGE_INSURANCE_TYPE(T.MISSING)',
       'MORTGAGE_INSURANCE_TYPE(T.OTHER)', 'PROP_TYPE(T.OTHER)',
       'PROP_TYPE(T.PU)', 'PROP_TYPE(T.SF)', 'OCCUPANCY_TYPE(T.OTHER)',
       'OCCUPANCY_TYPE(T.P)', 'FIRST_TIME_HOME_BUYER(N):BORROWER_CREDIT
_SCORE',
       'FIRST_TIME_HOME_BUYER(OTHER):BORROWER_CREDIT_SCORE',
       'FIRST_TIME_HOME_BUYER(Y):BORROWER_CREDIT_SCORE'],
      dtype='object')
```

Let's go ahead and transform our validation set.

In [43]:

```
X_tree_validation = df_validation[continuous_features_trees + cat_ordinal_features_
trees]
y_tree_validation = df_validation['DEFAULT_FLAG']

X_tree_validation_patsy = build_design_matrices([X_tree_train_patsy.design_info], d
f_validation, return_type="dataframe")[0]

X_tree_validation = pd.concat([X_tree_validation, X_tree_validation_patsy], axis=1)

X_tree_validation[cat_ordinal_features_trees] = ordinal_encoder.transform(X_tree_va
lidation[cat_ordinal_features_trees])

X_tree_validation.columns = X_tree_validation.columns.str.replace('[', '(').str.rep
lace(']', ')')
```

Now we are ready to start training trees.

Decision Tree

%%time dt_model = DecisionTreeClassifier(max_depth=30, min_samples_split=25, max_features=.5, min_impurity_decrease=.00001, random_state=201) dt_model.fit(X_tree_train, y_tree_train)

Let's plot our tree.

plt.figure(figsize=(20,20)) plot_tree(dt_model, feature_names=X_tree_train.columns, filled=True, fontsize=12) plt.show()

We can also look at the feature importance.

pd.DataFrame({'Importance': dt_model.feature_importances_}, index=X_tree_train.columns).sort_values(['Importance'], ascending=False)

Now we can predict on our validation set.

dt_pred = dt_model.predict_proba(X_tree_validation)[:,1]skill_score(df_validation['DEFAULT_FLAG'], dt_pred)

Let's look at the ROC curve:

plot_roc_curve(dt_model, X_tree_validation, df_validation['DEFAULT_FLAG'])

Random forest

A random forest and the boosted trees may take a long time to run, particularly on the full dataset, so you should be careful what you run on the full dataset. If something is taking forever, you can hit the stop button in the toolbar or go to "Kernel -> Interrupt Kernel".

%%time rf_model = RandomForestClassifier(n_estimators=500, max_features=.2, max_depth=30, min_samples_split=25, min_impurity_decrease=.00001, random_state=201, n_jobs=num_cpus) rf_model.fit(X_tree_train, y_tree_train)

We can also look at the feature importance.

pd.DataFrame({'Importance': rf_model.feature_importances_}, index=X_tree_train.columns).sort_values(['Importance'], ascending=False)

Now we can predict on our validation set.

rf_pred = rf_model.predict_proba(X_tree_validation)[:,1]skill_score(df_validation['DEFAULT_FLAG'], rf_pred)

Let's look at the ROC curve:

plot_roc_curve(rf_model, X_tree_validation, df_validation['DEFAULT_FLAG'])

Boosted trees model

Let's look at a boosted trees model.

In [ ]:

```
#Adding grid search parameters

parameters = {'max_depth': [4],
              'n_estimators': [130, 135, 140],
              'learning_rate': [.13, .14, .15, 16,]}
```

Search 1: parameters = {'max_depth': [5,15,25], 'n_estimators': [25,75,150], 'learning_rate': [0.001, .01, .1]} Best: {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 150} Search 2 parameters = {'max_depth': [4,6,8,10], 'n_estimators': [125, 150, 175], 'learning_rate': [.05, .1, .15]} {'learning_rate': 0.15, 'max_depth': 4, 'n_estimators': 150} Search 3 parameters = {'max_depth': [4,5,6], 'n_estimators': [140, 150, 160,170], 'learning_rate': [.1, .15, .175,.2]} {'learning_rate': 0.15, 'max_depth': 4, 'n_estimators': 140} Search 4 parameters = {'max_depth': [4],

'n_estimators': [130, 135, 140], 'learning_rate': [.13, .14, .15, 16,]} {'learning_rate': 0.14, 'max_depth': 4, 'n_estimators': 140}

In [ ]:

```
#Create parameter grid
from sklearn.model_selection import ParameterGrid
list(ParameterGrid(parameters))
```

In [ ]:

```
boost_grid_model = GridSearchCV(XGBClassifier(ccp_alpha=.0001,random_state=201), pa
ram_grid=parameters, cv=4, n_jobs=num_cpus, scoring='neg_brier_score')
```

In [ ]:

```
%%time
boost_grid_model.fit(X_tree_train, y_tree_train)
```

In [ ]:

```
boost_grid_model.best_params_
```

In [ ]:

```
boost_grid_predict = boost_grid_model.predict_proba(X_tree_validation)[:,1]
```

In [ ]:

```
skill_score(df_validation['DEFAULT_FLAG'], boost_grid_predict)
```

In [ ]:

```
#optimized model using the grid search features
```

In [46]:

```
%%time
boost_optimized = XGBClassifier(max_depth=4,
                          n_estimators = 140,
                          learning_rate=.14,
                          random_state=201,
                          ccp_alpha=.0001,
                          n_jobs=num_cpus)
boost_optimized.fit(X_tree_train, y_tree_train)
```

```
CPU times: user 14min 29s, sys: 1.42 s, total: 14min 31s
Wall time: 2min 28s
```

Out[46]:

```
XGBClassifier(ccp_alpha=0.0001, learning_rate=0.14, max_depth=4,
              n_estimators=140, n_jobs=12, random_state=201)
```

In [47]:

```
boost_optimized_predict = boost_optimized.predict_proba(X_tree_validation)[:,1]
```

In [48]:

```
skill_score(df_validation['DEFAULT_FLAG'], boost_optimized_predict)
```

Out[48]:

```
0.044922799597829036
```

In [49]:

```
pd.DataFrame({'Importance': boost_optimized.feature_importances_}, index=X_tree_tra
in.columns).sort_values(['Importance'], ascending=False)
```

Out[49]:

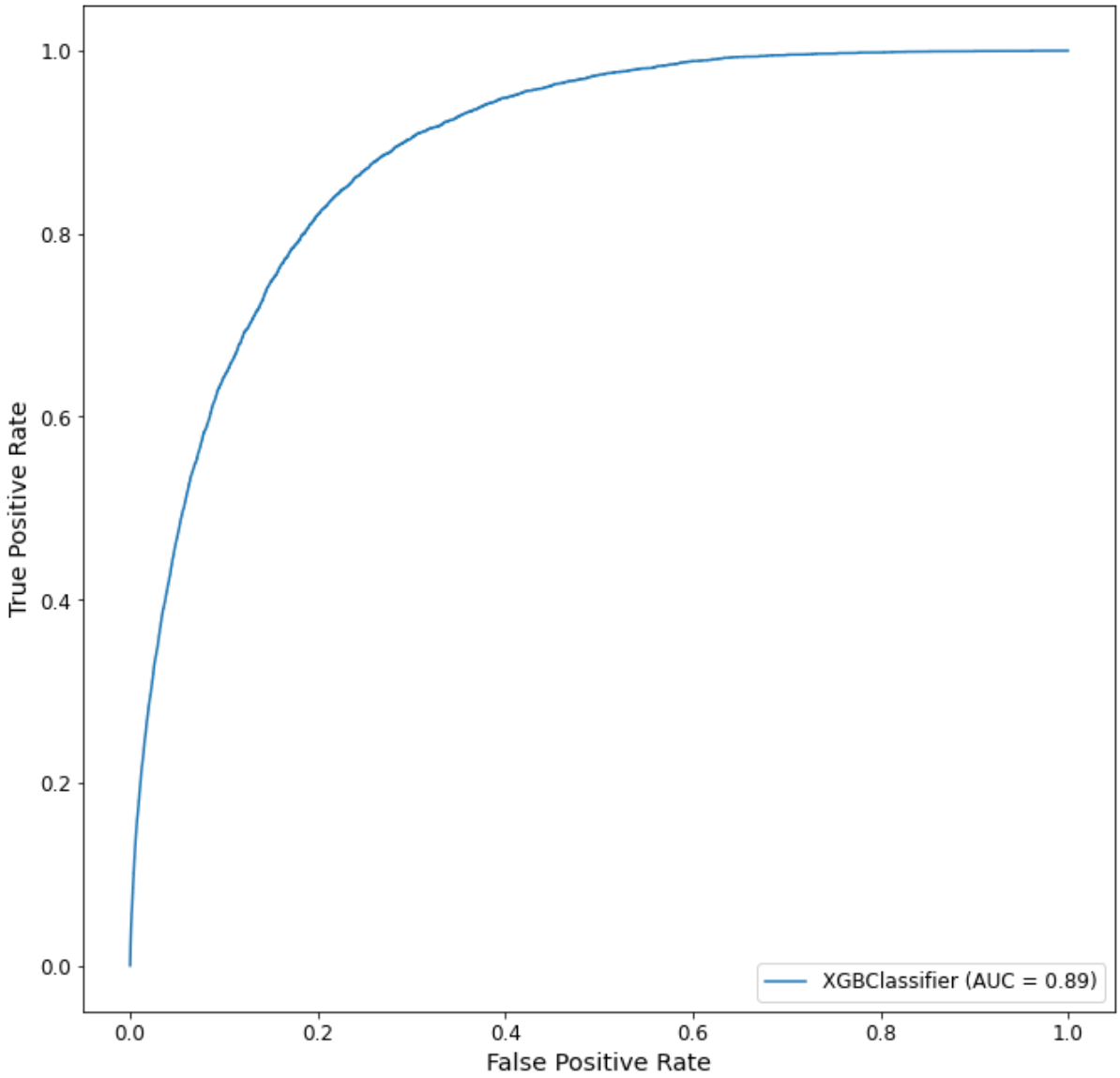
	Importance
BORROWER_CREDIT_SCORE	0.155351
ORIGINAL_INTEREST_RATE	0.125538
NUMBER_OF_BORROWERS	0.103399
MORTGAGE_INSURANCE_TYPE(T.MISSING)	0.093100
COBORROWER_CREDIT_SCORE	0.070975
LOG_OV	0.066324
OCCUPANCY_TYPE(T.P)	0.060595
ORIGINAL_COMBINED_LTV	0.060310
STATE	0.058715
PROP_TYPE(T.OTHER)	0.036245
OCCUPANCY_TYPE(T.OTHER)	0.035152
ORIGINAL_DTI	0.025801
PROP_TYPE(T.PU)	0.025081
PROP_TYPE(T.SF)	0.024492
FIRST_TIME_HOME_BUYER(Y):BORROWER_CREDIT_SCORE	0.023815
MORTGAGE_INSURANCE_TYPE(T.2.0)	0.010392
FIRST_TIME_HOME_BUYER(N):BORROWER_CREDIT_SCORE	0.010150
FIRST_TIME_HOME_BUYER(Y)	0.009484
FIRST_TIME_HOME_BUYER(OTHER):BORROWER_CREDIT_SCORE	0.005081
MORTGAGE_INSURANCE_TYPE(T.OTHER)	0.000000
FIRST_TIME_HOME_BUYER(OTHER)	0.000000
FIRST_TIME_HOME_BUYER(N)	0.000000

In [50]:

```
plot_roc_curve(boost_optimized, X_tree_validation, df_validation['DEFAULT_FLAG'])
```

Out[50]:

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x7f580d0a6c90>
```



In [ ]:

```
#Original code
```

In [ ]:

```
%%time
xgb_model = XGBClassifier(max_depth=5,
                          n_estimators = 150,
                          learning_rate=.1,
                          random_state=201,
                          n_jobs=num_cpus)
xgb_model.fit(X_tree_train, y_tree_train)
```

pd.DataFrame({'Importance': xgb_model.feature*importances*}, index=X_tree_train.columns).sort_values(['Importance'], ascending=False)We can also look at the feature importance.

In [ ]:

```
pd.DataFrame({'Importance': xgb_model.feature_importances_}, index=X_tree_train.col
umns).sort_values(['Importance'], ascending=False)
```

Now we can predict on our validation set.

In [ ]:

```
xgb_pred = xgb_model.predict_proba(X_tree_validation)[:,1]
```

In [ ]:

```
skill_score(df_validation['DEFAULT_FLAG'], xgb_pred)
```

In [ ]:

```
#default skill score: 0.02663355202801136
```

Let's look at the ROC curve:

In [ ]:

```
plot_roc_curve(xgb_model, X_tree_validation, df_validation['DEFAULT_FLAG'])
```

Neural Networks

Neural networks are like tree based models in that they will automatically construct useful interactions through the hidden layers, but they are also like linear models in that order of inputs matter. Therefore, for neural networks it is best to create dummy variables for all the categorical variables and to standardize the continuous variables.

To transform the data, we will use `patsy` again. And the formula will look similar to the linear model formula.

formula_nn = "DEFAULT_FLAG ~ 0 + standardize(ORIGINAL_LTV) + standardize(ORIGINAL_DTI) + standardize(BORROWER_CREDIT_SCORE) + FIRST_TIME_HOME_BUYER + STATE + standardize(YEAR)"formula_nn

Note that we do not include any interaction terms. You are free to do so, but neural networks can do a pretty good job of learning interactions. You may find that providing some pre-created interaction variables may still help, though.

We also, have `0 +` in the formula to keep the data from having an intercept term. Neural networks do not need intercepts.

Now we can build our X and y to train our model.

y_nn_train, X_nn_train = dmatrices(formula_nn, df_smaller_train, return_type="dataframe")

Let's look at what it made.

y_nn_trainX_nn_train

We will go ahead and build the validation matrix. In order to test on our validation set, we have to perform the *exact same transformations on it*. First, we will use our `build_design_matrices` function to get our `X_nn_validation` matrix. Remember the `y` values for the validation set are just `df_validation['IsBadBuy']`.

X_nn_validation = build_design_matrices([X_nn_train.design_info], df_validation, return_type="dataframe")[0]

Now we can predict on the validation set after we have built out model.

Let's build a `MLPClassifier`.

%%time nn_model = MLPClassifier(activation='relu', hidden_layer_sizes=(4,3), random_state=23) nn_model.fit(X_nn_train, y_nn_train)

And we can get our test predictions:

nn_pred = nn_model.predict_proba(X_nn_validation)[:,1]

And our score:

skill_score(df_validation['DEFAULT_FLAG'], nn_pred)

Let's look at the ROC curve:

plot_roc_curve(nn_model, X_nn_validation, df_validation['DEFAULT_FLAG'])

Visualize your predictions

No we will try to get a sense visually for what our predictions are doing. To choose what predictions and realizations to look at, just change the variables in the below cell. The plots will then reflect the new values next time you run their cells.

predictions = nn_pred realizations = df_validation['DEFAULT_FLAG']

First we will look at a calibration plot. On the x-axis are binned predictions made (probabilities) and on the y-axis are default rates (average of DEFAULT_FLAG) for those predictions. Where should the points lie if your predictions are perfectly calibrated?

predictions = pd.Series(predictions) realizations = realizations.reset_index(drop=True) bins = list(np.arange(-.01, .16, .01)) bin_indices = [] for i in range(len(bins) - 1): bin_indices.append(predictions[(predictions > bins[i-1]) & (predictions <= bins[i])].index.to_numpy()) x_values = [] y_values = [] for i in range(len(bins) - 1): x_values.append(np.mean(predictions[bin_indices[i]])) y_values.append(np.mean(realizations[bin_indices[i]])) plt.figure(figsize=(6,6)) plt.scatter(x_values, y_values) plt.xlabel('Predictions made within a bin') plt.ylabel('Default

rate for predictions made within a bin') plt.title('Calibration Plot') plt.xlim(0,0.16) plt.ylim(0,0.16) plt.plot([0,1], [0,1], color='r', linestyle='-', linewidth=2) plt.show()

Now we will look at discrimination plots: (1) the histogram of your predictions when defaults did not occurre (first histogram below) and (2) the histogram of your predictions when defaults did occur (second histogram below). Which histogram should have more mass closer to one? Which closer to zero? This is really about the accuracy of your predictions.

index_DEFAULT = realizations[realizations == 1].index.to_numpy() index_NO_DEFAULT = realizations[realizations == 0].index.to_numpy() plt.figure(figsize=(10, 6)) plt.subplot(121) # 121 means 1 row and 2 columns of plots and this is the first subplot predictions[index_NO_DEFAULT].hist(bins=20, density=1) plt.xlim(0, 0.2) plt.xlabel('Predictions made within a bin') plt.ylabel('Density of defaults within a bin') plt.title('Discrimination Plot (Non-Defaulted Loans Only)', fontsize=12) plt.subplot(122) # check_predictions[index_DEFAULT].hist(bins=20) predictions[index_DEFAULT].hist(bins=20, density=1) plt.xlim(0, 0.2) plt.xlabel('Predictions made within a bin') plt.ylabel('Density of defaults within a bin') plt.title('Discrimination Plot (Defaulted Loans Only)', fontsize=12) plt.show()

Predict on the Test Set

After you have gone through and chosen your specific model and the parameters from the model using your validation set, it is now time to go back and redo everything on the full training set in order to make predictions on the test set. Note that the below will assume you want to make the same choices around formatting data as you did for the `df_smaller_train` set above. The below is really just going through the above steps to create the data but replacing `df_smaller_train` with `df_train` and `df_validation` with `df_test`.

First, we will refit the imputers and impute on `df_train` and `df_test`.

In [51]:

```
imputer_mean_final = SimpleImputer(missing_values=np.nan, strategy='mean')
imputer_zero_final = SimpleImputer(missing_values=np.nan, strategy='constant', fill
_value=0)
categorical_imputer_final = CategoricalImputer(other_threshold=.01)
```

In [52]:

```
imputer_mean_final.fit(df_train[continuous_mean])
df_train[continuous_mean] = imputer_mean_final.transform(df_train[continuous_mean])
df_test[continuous_mean] = imputer_mean_final.transform(df_test[continuous_mean])
```

In [53]:

```
imputer_zero_final.fit(df_train[continuous_zero])
df_train[continuous_zero] = imputer_zero_final.transform(df_train[continuous_zero])
df_test[continuous_zero] = imputer_zero_final.transform(df_test[continuous_zero])
```

In [54]:

```
categorical_imputer_final.fit(df_train[categorical_variables])
df_train[categorical_variables] = categorical_imputer_final.transform(df_train[cate
gorical_variables])
df_test[categorical_variables] = categorical_imputer_final.transform(df_test[catego
rical_variables])
```

Now, we will re-create the logistic regression data necessary to train and predict. This will use the same `formula_logit` you created when you were first training the logistic regression.

y_logit_train_final, X_logit_train_final = dmatrices(formula_logit, df_train, return_type="dataframe")

Now, we can retrain the logistic regression model on the new data set. NOTE: You should adjust the parameters values to what you chose above.

%%time rlr_final = LogisticRegression(C=1, random_state=201) rlr_final.fit(X_logit_train_final, y_logit_train_final)

Now we can build the `X_test` matrix and get final predictions from the logistic regression.

X_test = build_design_matrices([X_logit_train_final.design_info], df_test, return_type="dataframe")[0]

rlr_pred_final = rlr_final.predict_proba(X_test)[:,1]

Now, we will recreate the tree based models data set.

In [55]:

```
X_tree_train_final = df_train[continuous_features_trees + cat_ordinal_features_tree
s]
y_tree_train_final = df_train['DEFAULT_FLAG']
```

If you created dummy variables or interactions, you will want to run the next line. If not, you can skip it. This will use the same `formula_tree` you made when first setting up the data set for trees.

In [56]:

```
X_tree_train_patsy_final = dmatrix(formula_tree, df_train, return_type="dataframe")
X_tree_train_final = pd.concat([X_tree_train_final, X_tree_train_patsy_final], axis
=1)
```

Now we ordinal encode variables. If you did not choose to ordinal encode any variables, you can skip this.

In [57]:

```
ordinal_encoder_final = OrdinalEncoder()
ordinal_encoder_final.fit(X_tree_train_final[cat_ordinal_features_trees])
X_tree_train_final[cat_ordinal_features_trees] = ordinal_encoder_final.transform(X_
tree_train_final[cat_ordinal_features_trees])
```

Finally, we have to replace the characters that xgboost doesn't like.

In [58]:

```
X_tree_train_final.columns = X_tree_train_final.columns.str.replace('[', '(').str.r
eplace(']', ')')
```

Now, we need to do these steps for the test set.

In [59]:

```
X_tree_test = df_test[continuous_features_trees + cat_ordinal_features_trees]
y_tree_test = df_test['DEFAULT_FLAG']

X_tree_test_patsy = build_design_matrices([X_tree_train_patsy_final.design_info], d
f_test, return_type="dataframe")[0]

X_tree_test = pd.concat([X_tree_test, X_tree_test_patsy], axis=1)

X_tree_test[cat_ordinal_features_trees] = ordinal_encoder_final.transform(X_tree_te
st[cat_ordinal_features_trees])

X_tree_test.columns = X_tree_test.columns.str.replace('[', '(').str.replace(']',
')')
```

Now we can retrain our tree based models, and make final predictions. Note you should adjust parameters to whatever you found above, and you may not need to run all of the below code if you don't intend to use some of the models.

First, the decision tree.

%%time dt_model_final = DecisionTreeClassifier(max_depth=30, min_samples_split=25, max_features=.5, min_impurity_decrease=.00001, random_state=201) dt_model_final.fit(X_tree_train_final, y_tree_train_final) dt_pred_final = dt_model_final.predict_proba(X_tree_test)[:,1]

Second, the random forest.

%%time rf_model_final = RandomForestClassifier(n_estimators=500, max_features=.2, max_depth=30, min_samples_split=25, min_impurity_decrease=.00001, random_state=201, n_jobs=num_cpus) rf_model_final.fit(X_tree_train_final, y_tree_train_final) rf_pred_final = rf_model_final.predict_proba(X_tree_test)[:,1]

Finally, the boosted tree.

In [ ]:

```
#{'learning_rate': 0.14, 'max_depth': 4, 'n_estimators': 140}
```

In [60]:

```
%%time
xgb_model_final = XGBClassifier(max_depth=4,
                          n_estimators = 140,
                          learning_rate=.14,
                          random_state=201,
                          n_jobs=num_cpus)
xgb_model_final.fit(X_tree_train_final, y_tree_train_final)
xgb_pred_final = xgb_model_final.predict_proba(X_tree_test)[:,1]
```

```
CPU times: user 15min 48s, sys: 2.15 s, total: 15min 50s
Wall time: 2min 2s
```

Now we can create the data set necessary for the neural network and train that as well.

y_nn_train_final, X_nn_train_final = dmatrices(formula_nn, df_train, return_type="dataframe")X_nn_test = build_design_matrices([X_nn_train_final.design_info], df_test, return_type="dataframe")[0]

We can now train our final neural network model. Remember to use the same parameters that you used when validating the model.

%%time nn_model_final = MLPClassifier(activation='relu', hidden_layer_sizes=(4,3), random_state=23) nn_model_final.fit(X_nn_train_final, y_nn_train_final)

And predict on the test set:

nn_pred_final = nn_model_final.predict_proba(X_nn_test)[:,1]

Now you can choose to ensemble however you would like. Below is one possibility of just ensembling all of the above models.

In [ ]:

```

```

In [61]:

```
#final model no ensembling
final_pred = xgb_pred_final
```

Now, see you skill score on your final prediction.

In [62]:

```
skill_score(df_test['DEFAULT_FLAG'], final_pred)
```

Out[62]:

```
0.04610353306459476
```

Set a Threshold and Compute Revenue

Ultimately, we care about using the predictions of the model to decide whether or not to accept a mortgage application. Therefore, we have to decide what we consider to be too high of a probability of default before we reject a loan application. However, if we reject a loan application that does not default, we lose some potential revenue.

We will load in a fresh copy of the data to ensure we are computing the correct revenue.

%%time if not full_data_set: df_test_fresh = pd.read_csv('../Shared Data (Read Only)/Fannie Mae Data/FannieMaeSmallTest.csv', sep='|') elif full_data_set: df_test_fresh = pd.read_csv('../Shared Data (Read Only)/Fannie Mae Data/FannieMaeTest.csv', sep='|')

One way of doing this is by setting a threshold where if the probability of default is lower than the threshold, accept the loan. Therefore, we can set the threshold either too high or too low. If it is too high, we lose out on potential revenue from accepting good loans, but if it too low, we take losses on loans that do default.

The below function will take in your predictions and a threshold, and it will produce a list of `True` and `False` values where `True` means you accept the loan and `False` means you do not accept the loan.

def accept_by_threshold(predictions, threshold): return predictions <= threshold

You can use it like below:

loan_decisions = accept_by_threshold(final_pred, .05)

You can see that we end up accepting about 95% of the loans if we accept a 5% chance of default according to our `final_pred` .

loan_decisions.mean()

The `accept_by_threshold` is one way to make this decision, and you are encouraged to think about other ways to do this.

The below function will take in your loan decisions and compute the revenue (in millions) from the accepted loans.

def loan_loss_revenue(loans_accepted): revenue = (df_test_fresh[loan_decisions]['ORIGINAL_VALUE']*.005).sum() - df_test_fresh[loan_decisions]['NET_LOSS'].sum() return revenue/1000000

The revenue generated per loan is set at .5% of the original loan value. This is taken from the Quarterly Mortgage Bankers Performance Report, Q1 2018, where the average revenue per loan was about \$1500. Since the average loan size in the Fannie Mae data set for 2018 was \\$330,000, this amounts to roughly .5% of the original loan value as revenue for the average mortgage banker in this time period. So, if you accept a loan (whether or not the loan ultimately defaults), the `loan_loss_revenue` will add .5% of the loan value for that loan.

However, if you accept a loan that does default, then the `NET_LOSS` of that loan will be subtracted from your total revenue.

This function allows you to compute the expected revenue from accepting your recommendations. You can use it as below.

```
loan_loss_revenue(loan_decisions)
```

Write out the data with your predictions

It is often helpful to take our predictions after we have made them, and visualize our errors to get a sense for how we might want to change and improve our model. Here we will append our predictions to the original testing set, and then we will write out the testing set with our predictions appended so that you can download the csv file and use Tableau to iterate on your model.

You will need the `df_test_fresh` data set loaded in the computing revenue section of this notebook. Once we have that you can add your predictions.

```
df_test_fresh['PREDICTIONS_DEFAULT'] = final_pred
```

Write out the csv of the data with your predictions. Use this csv in Tableau to study your predictions geographically and try to determine where you might be making errors.

```
df_test_fresh.to_csv('FannieMaeTestWithPredictionsDefault.csv', sep='|', index=False)
```