Statistical Learning on Loan Decision

Author:
Ching-Yi Wang (Ian)

Date: September 14, 2023

1 Introduction

Library: tidymodels, tidyverse, kernlab, xgboost

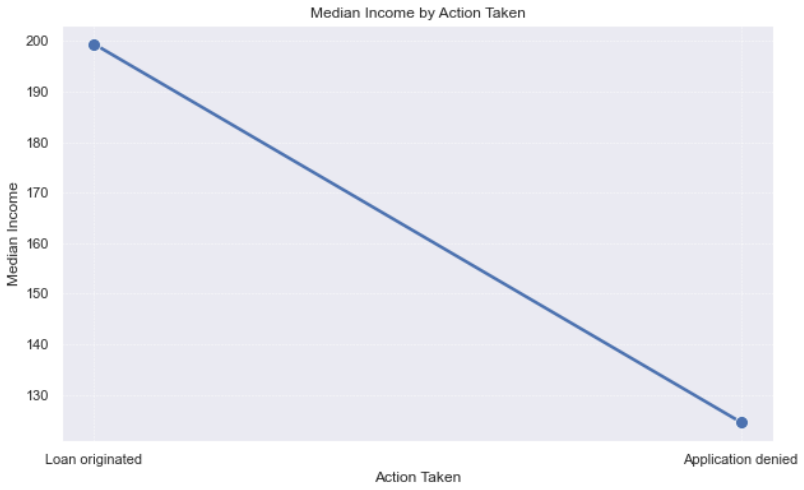
Our classification project focuses on predicting loan application outcomes, specifically "Loan originated" or "Application denied," vital for efficient financial processing. We've analyzed diverse variables, including loan type, purpose, demographics, credit scores, and loan amounts. This data-driven journey aims to uncover key factors influencing loan outcomes, streamlining financial decisions and enhancing inclusivity. We believe credit score and loan amount significantly associate with the response variable. Higher credit scores signal responsible financial behavior and enhance approval chances, while lower scores heighten risk and may lead to denials. Larger loan requests amplify lender risk, triggering stringent scrutiny, affecting approval likelihood. This report offers insights for financial stakeholders, promoting efficient lending practices.

Reference:

1. XGBoost Documentation.
   https://xgboost.readthedocs.io/en/stable
2. A Guide on XGBoost Hyperparameters Tuning.
   https://www.kaggle.com/code/prashant111/a-guide-on-xgboost-hyperparameters-tuning
3. Tidymodels Stacks.
   https://stacks.tidymodels.org
4. Tidymodels Tune.
   https://tune.tidymodels.org
5. Tidymodels Parsnip.
   https://parsnip.tidymodels.org
6. Linear support vector machines (SVMs) via kernlab
   https://parsnip.tidymodels.org/reference/details_svm_linear_kernlab

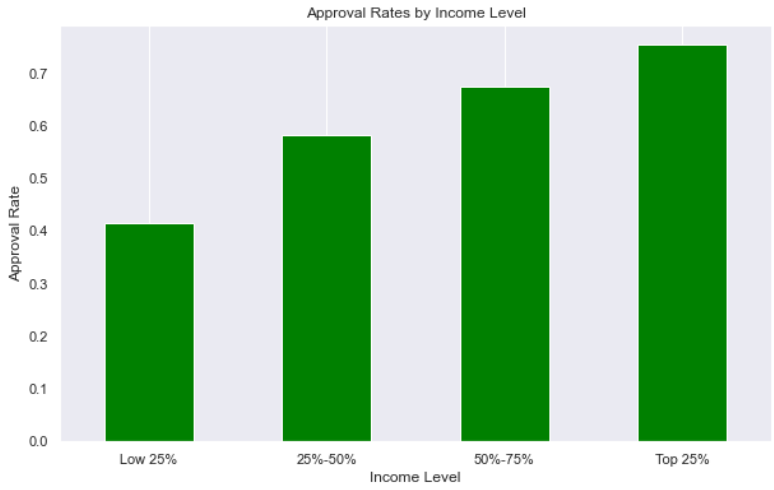
2 Exploratory Data Analysis

Figure 1: Line Plot: Income vs. Action Taken



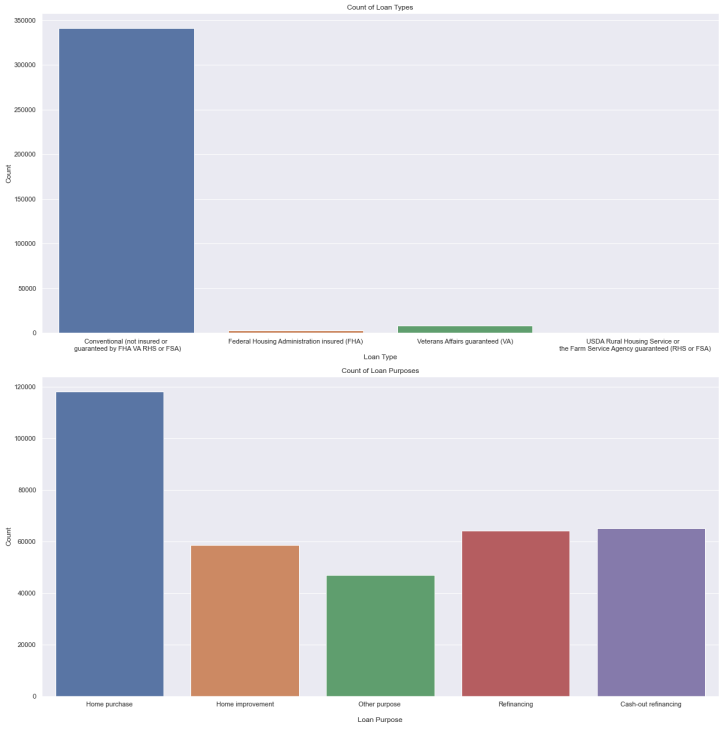
Summary: According to the line plot of Median Income vs Action Taken, there is evidence to support the median income is associated with the loan action taken. The median income difference between loan application approval and denial is 75 units which is considered as a large gap between two conditions. Therefore, the income can be a main factor that impacts the loan action taken.

Figure 2: Barplot: Income vs. Action Taken



Sumary: Based on this graph, it's evident that when the income increase, the approval rate will also increase. By sampling the income from low 25%,$25\% - 50\%, 50\% - 75\%$, top 25%, we can see the top 25 % will has 30 % more chance to getting a loan approval compare to the lower 25%. Therefore, this indicate the income is significant to the loan approval rate.

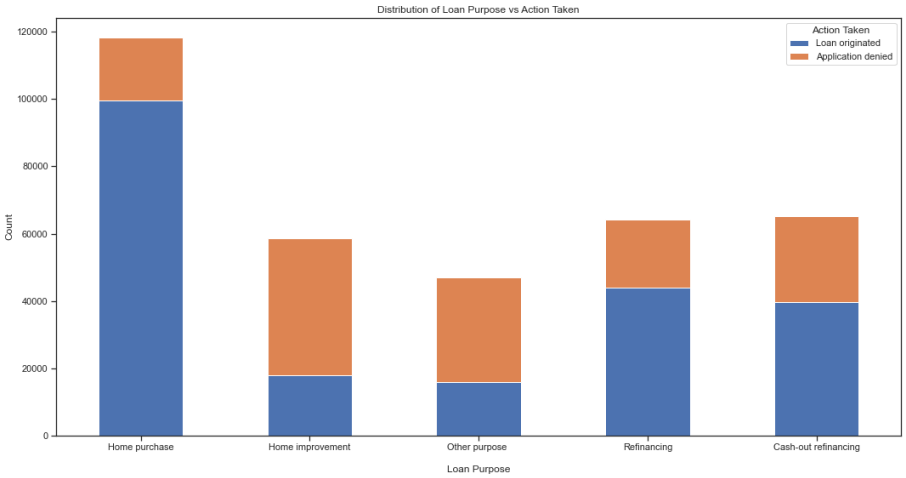
Figure 3: Count of Loan Types & Loan Purpose



Summary: The graph displays the count of different loan types in the dataset. Conventional loans have the highest count, exceeding 300,000, making them the most prevalent loan type. Veterans Affairs (VA) guaranteed loans follow as the second most common. Federal Housing Administration (FHA) insured loans come third in frequency, while loans guaranteed by the USDA Rural Housing Service or the Farm Service Agency (RHS or FSA) have the lowest count among the four types.

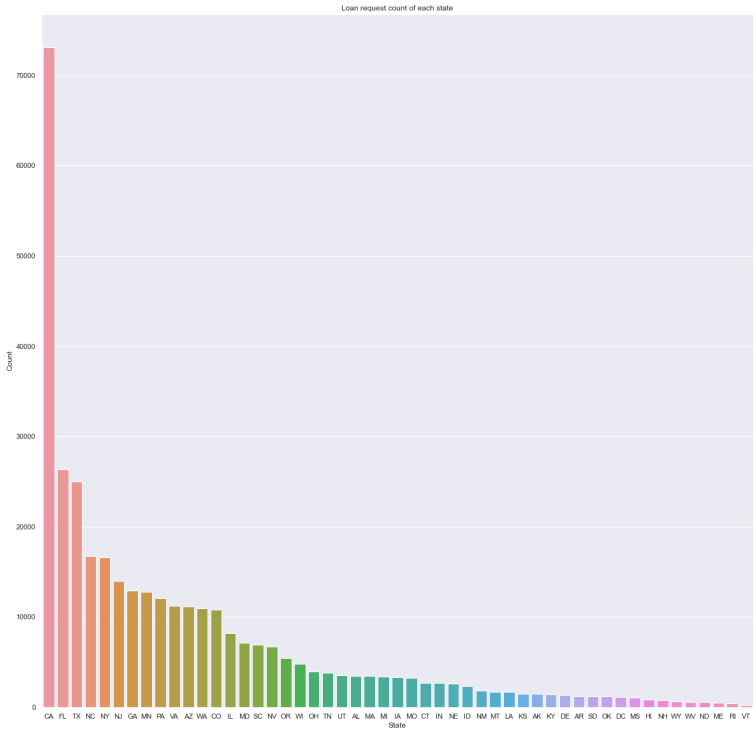
The dataset's loan purpose distribution shows that "Home purchase" loans are the most common, followed by "Cash-out refinancing" and "Refinancing" loans. "Home improvement" loans rank fourth, while "Other purpose" loans are less common.

Figure 4: Barplot: Probability of Action Taken by Loan Purpose



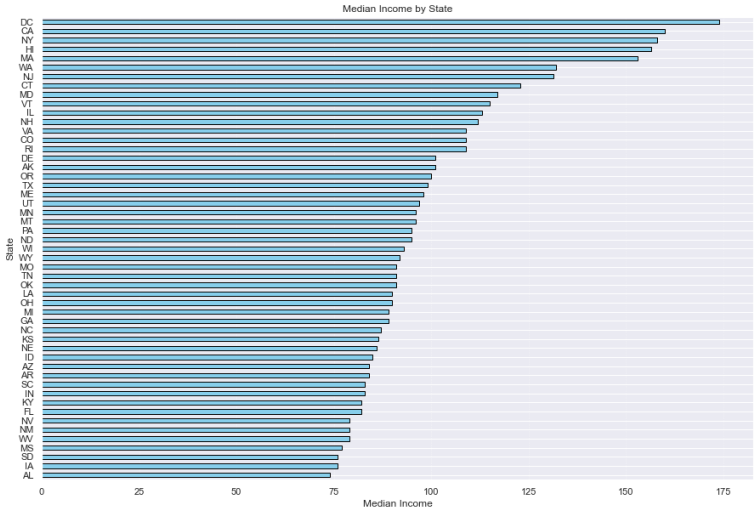
Summary: From the bar plot of Loan Purpose vs Action Taken, it's evident that the purpose of the loan plays a significant role in its approval rate, the home purchase has the highest loan approved rate and the home improvement seems to have the highest application denied rate. Therefore, people have the house purchase as their loan purpose seems to be more likely to be approved. However, the loan purpose and action taken might also be impacted by multiple factors such as income and loan amount.

Figure 5: Barplot: State Loan Request Count



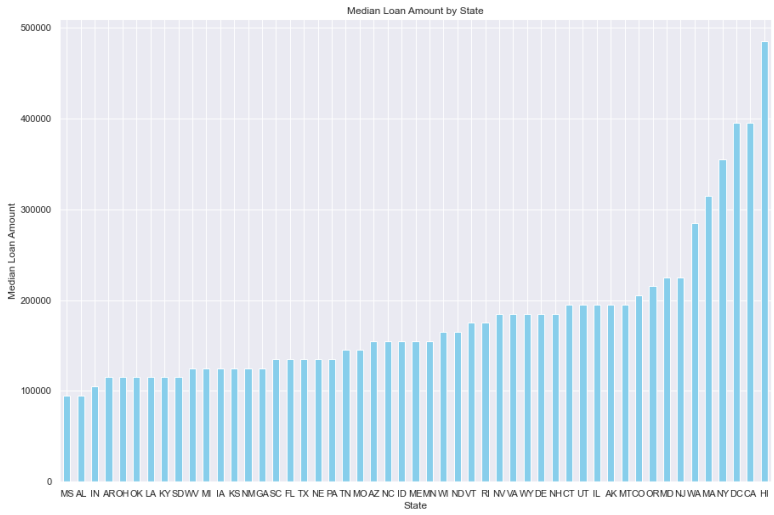
Summary: The graph reveals that California has the highest loan request count, exceeding 70,000, followed by Florida and Texas. Conversely, Vermont and Rhode Island have the lowest request counts among all the states. This pattern suggests a correlation between a state's population size and its loan request volume, with larger states having higher request counts.

Figure 6: Barplot: Median Income vs States



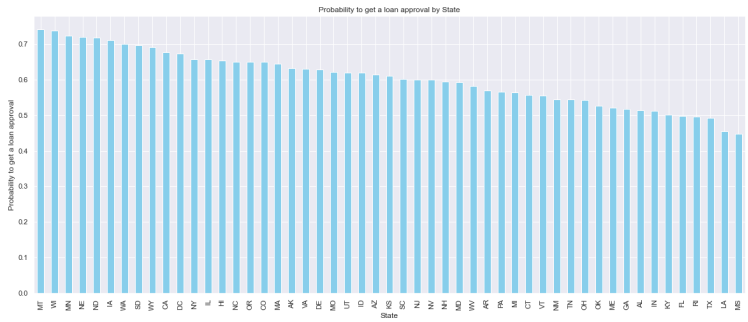
Summary: By observing the median income by state, the plot indicates the cities that have relatively high income are "Washington DC, California, New York, Hawaii, and Massachusetts." This might be due to their urbanization which often comes with a higher living expense and the urbanization level is also an important factor to indicate the concentration of industries which could also lead to a higher income.

Figure 7: Bar Plot: Median Loan amount vs States



Summary: Combining the median income by state and the median loan amount by state indicates the lower median income states will have relatively low loan amounts and the higher median income states will have relatively high loan amounts. Since the income and loan amount represent a positive correlation, the loan approval might not be impacted due to the different loan amount. However, the loan amount might also rely heavily on the loan purpose, loan type, and the credit, where the loan amount could have a certain level of positive relationship with these factors.

Figure 8: Probability of loan approval by states



Summary: According to the loan approval probability vs. states plot, there is an observation which indicates the highest loan approval probability is around 75% and the lowest is around 45% which indicates a large gap in chance to get loan approval between the states and the largest difference is around 30% which is between Montana and Mississippi. Combining the graph above, the states with income seem to be related to the probability of getting a loan approval, where the loan approval chance would increase as the income increases which indicates a positive relationship between income and probability of loan approval.

3 Preprocessing / Recipe

Summary: Our first course of action after loading the "train2.csv" as **train** was changing our variable of interest, **action_taken**, to a factor. There were numerous variables in the training and testing sets that contained significant amounts of NA values. In order to identify these columns we distinguished columns containing NA values (**na_cols**) from those columns with only NA values (**na_only)**. From the subset **na_cols** we used the **replace_na()** function along with the information regarding NA values for each variable from the **metadata.csv** file. These NA values commonly were meant to be represented by numeric entries such as 4, 7, 6, or "8888". The same process conducted for cleaning up the **train** data was performed on the "test2.csv", our **test** data set.

We then updated the **na_only** columns of the train data so that all the NA values were replaced with a 0. We decided to use a boosted tree which randomly selected 40 variables at each split, containing 100 trees. We set other limits for this boosted tree such as the minimum number of data points for a split to 10 points, the maximum number of splits for a tree to 10, along with learning rate of 0.2, a loss reduction of 0.01, while exposing all data to the fitting.

For our recipe, we made our prediction of **action_taken** based on all of the non-character columns in **train**. We then removed the id variable, utilized the **step_impute_mean()** function for all numeric predictors, so missing values would be replaced by the mean. Next was to use **step_zv()** to remove all predictors which contain only 1 unique value. Using **step_center()** we centered all numeric data, and using **step_dummy()** we created dummy variables for all nominal predictors aside from the outcome variable. After these final steps we were able to build our workflow, and begin testing the effectiveness of different models.

4 Candidate models/Model evaluation/Tuning

Model Candidates:

For our initial selections of models, we started with a logistic regression model with the "glmnet" engine and a decision tree model, both of which had very few or no tunable parameters. We saw that the decision tree model outperformed the logistic regression model and decided to work on a boosted tree model with the "xgboost" engine and a random forest model to have more tunable parameters with decision trees. We also included a mlp (Multilayer Perceptron) model and a svm (Support Vector Machine) model. However, for both models we found that the long runtime caused by computation made it difficult to tune the models in a reasonable time and for overall efficiency. The next section will further discuss the model evaluation process through a stack and other methods to come to the conclusion that a boosted tree would be our ideal model for the data set. Below is a chart of the candidate models.

Model Type	Engine	Recipe	Hyperparamters	10-Fold Validation Score
Logistic Regression	glmnet	recipe(action_taken ~ ., data = train %>% select_if(~!is.character(.))) %>% step_rm(id) %>% step_impute_mean(all_numeric()) %>% step_zv(all_predictors()) %>% step_center(all_numeric()) %>% step_dummy(all_nominal(), -all_outcomes())	penalty = 0	0.9717
Decision Tree	rpart	rec <- recipe(action_taken ~ ., data=train_df) %>% step_impute_mean(all_numeric()) %>% step_zv(all_predictors()) %>% step_center(all_numeric()) %>% step_dummy(all_nominal(), -all_outcomes())	NA	0.975
Random Forest	ranger	rf_recipe <- recipe(action_taken ~ ., data = train %>% select_if(~!is.character(.))) %>% step_rm(id) %>% step_impute_mean(all_numeric()) %>% step_zv(all_predictors()) %>% step_center(all_numeric()) %>% step_dummy(all_nominal(), -all_outcomes())	trees = 50, min_n = 10	0.9843
Boosted Tree	xgboost	recipe(action_taken ~ ., data = train %>%	mtry = 40, trees = 100, min_n = 10, tree_depth = 10,	0.9853

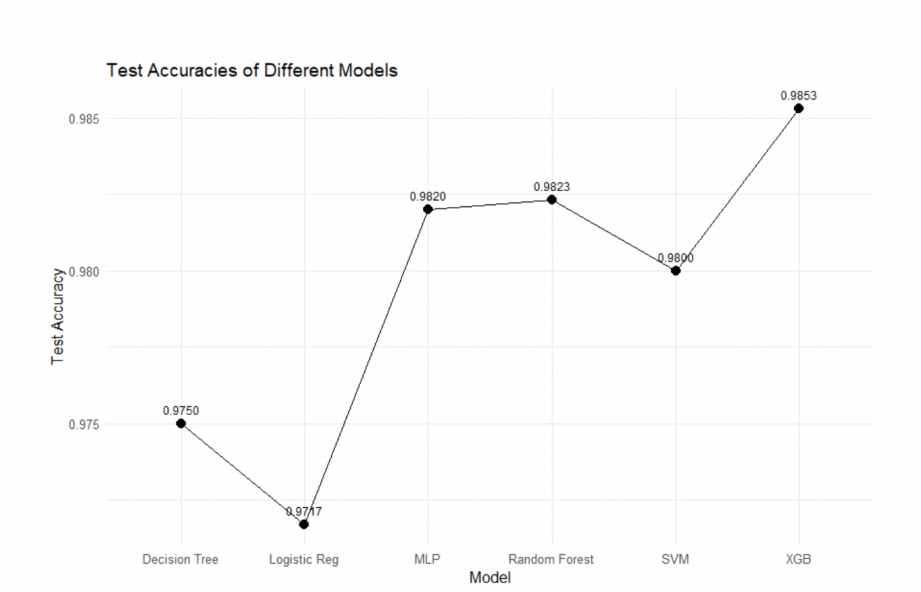
		select_if(~!is.character(.))) %>% step_rm(id) %>% step_impute_mean(all_numeric()) %>% step_zv(all_predictors()) %>% step_center(all_numeric()) %>% step_dummy(all_nominal(), -all_outcomes())	learn_rate = 0.2, loss_reduction = 0.01 sample_size = 1 stop_iter = 5	
MLP	keras	recipe(action_taken ~ ., data=train) %>% step_impute_mean(c("income", "combined_loan_to_value_ratio", "loan_term", "introductory_rate_period", "property_value", "multifamily_affordable_units")) %>% #knn taking so long step_zv(all_predictors()) %>% step_normalize(all_numeric()) %>% step_dummy(all_nominal(), -all_outcomes())	activation = "relu" learn_rate = 0.01 dropout = 0.2, hidden_units = 40 penalty = 0, epochs = 60	0.982
SVM	kernlab	recipe(action_taken ~ ., data = train %>% select_if(~!is.character(.))) %>% step_rm(id) %>% step_impute_mean(all_numeric()) %>% step_zv(all_predictors()) %>% step_center(all_numeric()) %>% step_dummy(all_nominal(), -all_outcomes())	cost = 1, margin = 0.1	0.980

5 Model Evaluation and Tuning

Trying out as many models as we wanted was not entirely possible in this competition. The vast amount of data meant that training models would often take several hours. Even before tuning our models, we figured that our choice of candidate models must be sensible. We did not use Latin Hypercube Sampling like we did last time. Instead, we manually tried tweaking a couple values with our previous knowledge from the class and regression competition, as well as using a stack to confirm our findings on our strong models.

Being a classification project, we first explored a very simple model, the logistic regression. As anticipated, it was the quickest to run, but giving an average score. One interesting thing we noticed was that tweaking the parameters had not changed the validation accuracy/score by a lot, or the Kaggle score. We discuss more about this later. We explored other models including decision tree, random forest, xgboost, MLP and SVM. The table below shows the score and average time for each model.

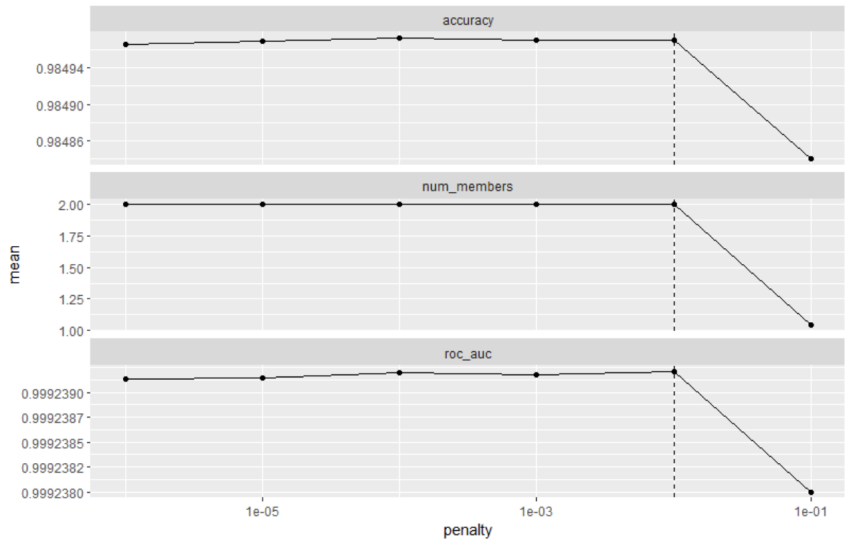
Model	Test Accuracy	Estimated time range to train	Important Notes
Logistic Regression	0.9717	~20-30 minutes	Tweaking parameters did not change accuracy by much
Decision Tree	0.9750	~35-45 minutes	Sometimes ran into new levels and NA issues
Random Forest	0.9823	~50-70 minutes	Tweaking parameters did not change accuracy by much
XGBoost	0.9853	~1.5-2.5 hours	Tweaking parameters showed good differences
MLP	0.9820	~3-8 hours	Time is very dependent on epoch iterations and k folds, leading to huge range of time taken to train (discussed below)
SVM	0.9800	~4-8 hours	Very time and resource intensive, no matter the parameters

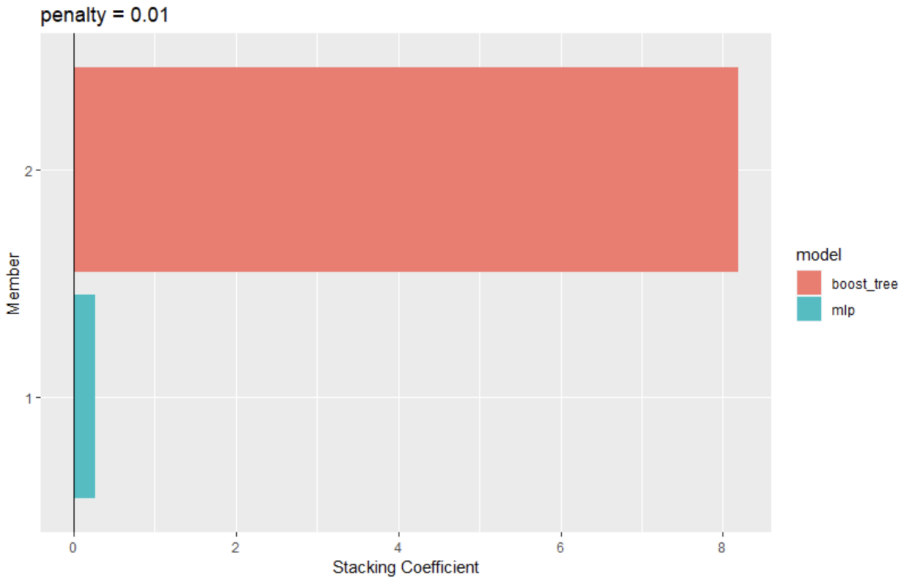


To train the MLP, the model would go through all the epochs for each k fold. And due to the large size of the dataset, on average 1 epoch took around 17-20 seconds. And so doing 200 epochs for 10 fold validation didn't make sense, since it would've taken over 11 hours. So we stuck with 50 epochs, and tried both 5 and 10 folds, both of which gave similar results, 10 having the better accuracy.

Ultimately, after seeing our candidate models, we decided to fit a stack onto them to get the best from each mode. Moreover, we wanted to see which models have the largest stacking coefficient. This is because a larger coefficient would mean it contributes more to the stack, and using this information, we can focus our time on tweaking important candidate models. Using control_stack_resamples() in our models, we could put together a stack. Slightly short on time, we decided two of the best candidate models and used them in the stack rather than all 6. We chose what we thought were the 2 best, XGBoost and MLP.

As we can see, these 2 models were within the top 3 models. Our objective with the stack was not only to see whether the stack as a whole performed better, but also to observe the important models within, and then tweak them individually. Some of the plots for the stack are:


accuracy
0.98494
0.98490
0.98486
num_members
2.00
1.75
1.50
1.25
1.00
mean
roc_auc
0.9992390
0.9992387
0.9992385
0.9992382
0.9992380
1e-05
1e-03
1e-01
penalty


penalty = 0.01
2
1
Member
0
2
4
6
8
Stacking Coefficient
model
boost_tree
mlp

The first autoplot suggests our best model had the penalty of 0.01. Focusing on the second plot, we can clearly see that the boost_tree (xgboost) model is dominating the stack, with a stacking coefficient of nearly 20 times that of the mlp model. This model had given us the highest public score (at the time) of 0.98824.

Leveraging the second plot, we decided to revisit the xgboost model to further tweak and see whether it gave a better score. Doing this, we found that increasing our trees from 40 to 100, and observed significant improvements to not only the test validation score, but also the public score as it rose to 0.9884. We didn't want to go higher because of resource constraints and primarily overfitting, since that did occur in the regression project.

We did try stacking xgboost, mlp and logistic regression, but the coefficients showed 100% xgboost, which could've been due to an error.

6 Discussion of Final Model

There were several reasons for choosing XGBoost and MLP for the stack, and not others. The reason we didn't choose Random Forest, regardless of the higher score, is because tweaking the parameters did not change the accuracy as much.

One crucial step was our preprocessing and feature engineering. Our recipes allowed us to better gauge the missing values and further increase interpretability for our models, which not only led to better results but also reduced computational time. Using the stack helped us ultimately choose our XGBoost model, since it showed us that XGBoost is the main reason for the higher score, and so we decided to further explore that. We know that stacks are typically more robust than individual models. If one model performs poorly on certain portions of the data, the other model may compensate for it. This can lead to a more stable and reliable prediction.

For the stack itself, we carefully chose XGBoost and MLP due to several reasons. Both of them are diverse model types. XGBoost is a gradient boosting algorithm, while MLP is a neural network. By combining these, we were hoping to capture a wider range of patterns and relationships within your data. Stacking allowed us to leverage the strengths of both XGBoost and MLP. XGBoost is known for its ability to handle structured data and gradient boosting, while MLP can capture complex, non-linear relationships. The combination can lead to improved predictive accuracy, especially for a large and complex data like ours. Overfitting is a common concern when using complex models like MLP. By combining MLP with XGBoost, we looked to reduce the risk of overfitting in the ensemble.

Some disadvantages mainly included complexity and resource intensive. Stacked ensembles add complexity to the modeling pipeline, having multiple models to train, tune, and manage. This made our workflow more challenging and resource-intensive. This is especially relevant given that our 2 models: XGBoost and MLP, are one of the most computationally expensive models. The training time also seemed to vary to which hardware our group had. Tuning hyperparameters for multiple models can be time-consuming, and was impossible in our case. Needing to optimize the parameters for XGBoost, MLP, was chal-

lenging to coordinate effectively. Combining complex models like XGBoost and MLP, the overall interpretability of our stack decreases. In the sense that it can be challenging to explain how the ensemble arrived at a particular prediction. As with all stacks, the overall performance of the stacked ensemble depends on the quality of predictions from individual base models. If one of the base models performed poorly, it may negatively impact the stack's performance.

As for our final model, that is just the XGBoost, was decided mainly from the stack, and the second autoplot showing the stacking coefficients. Seeing that the XGBoost model, primarily drove our powerful results, we decided to further explore and tweak parameters. Increasing the number of trees did the job, but we had to be mindful of not overfitting the data. Using our experience from the previous project, we decided to change from 40 to 100 trees. We didn't want to go over since it had a higher risk of overfitting.

The strengths of our XGBoost model include:

High Predictive Accuracy: XGBoost is well-suited for datasets with a large number of variables like ours. It excels at capturing complex, non-linear relationships between predictors and the target variable, which can lead to high predictive accuracy.

Parallel Processing: XGBoost can leverage parallel processing capabilities, making it faster to train than some other algorithms. This is advantageous when dealing with a large dataset.

Hyperparameter flexibility: You can fine-tune the model by adjusting hyperparameters to achieve optimal performance, which can be particularly useful when dealing with a dataset with many variables.

Some weaknesses include:

Hyperparameter Tuning: Tuning XGBoost's hyperparameters can be challenging and computationally expensive, given the size of the dataset and the large number of variables. Finding the optimal set of hyperparameters may require significant computational resources and time.

Interpretability: XGBoost is not highly interpretable. Understanding the precise reasons behind a particular prediction can be complex, especially with a large number of predictors.

Data Quality: XGBoost's performance is sensitive to the quality of the input data. Noisy data, present abundantly in our datasets, can lead to suboptimal results. Ensuring data quality through preprocessing and cleaning is essential in this case.

Some improvements could have been to use nested cross validation, though would be very time and resource consuming, could've given a better and realistic test score. We also should've worked efficiently, so it would allow us to try more combinations of stacks and parameters for models. We did carefully analyse the raw data, and preprocessed it effectively, but perhaps we could've done it better. One improvement that certainly would've improved our data is imputing missing values via KNN. We tried this multiple times, however, due to the large dimensions of our data, it took way too long to impute. This extra step of feature engineering would've greatly benefited our data. Specific variables like income, combined_loan_to_value_ratio, property_value and a few others, would have greatly improved if we successfully imputed through KNN. Due to their large variability, imputing the mean did not make sense since if a property value or income was very high, imputing the mean instead would severely skew that result. We could've also implemented robust outlier detection and handling techniques. Outliers can have a significant impact on model performance, and addressing them appropriately can improve results, especially on a MLP model.

Additional data, but not trivial like the type of purchaser, interest rate, debt to income ratio, credit history, would be useful. Information about the borrower's employment status and industry would provide good insight for income stability. Educational background can be indicative of earning potential and financial responsibility. The marital status and number of dependents on the borrower can be relevant in assessing financial stability and responsibilities. Other data providing financial stability would help like number of different accounts (savings, investment, etc), length of employment, etc.

7 R Code for Final Model

```r
# Load necessary libraries
library(tidymodels)
library(tidyverse)
library(kernlab)
library(xgboost)

# Read the training data from "train2.csv" and convert the "action_taken" column to a factor
train <- read_csv("train2.csv")
train$action_taken <- as.factor(train$action_taken)

# Identify columns with missing values
na_cols <- names(train[, colSums(is.na(train)) > 0])
na_only <- names(train[, colMeans(is.na(train)) == 1.0])

# Replace missing values with specific values for selected columns
train[na_cols] <- train[na_cols] %>%
  replace_na(list(
    ethnicity_of_applicant_or_borrower_1 = 4,
    ethnicity_of_applicant_or_borrower_2 = 4
    # ... (other columns)
  ))

# Select columns without missing values and replace missing values in the other columns with zeros
names(train[, colSums(is.na(train)) > 0])
train_full <- train %>% select(!na_only)
train_zero <- train %>%
  select(na_only) %>%
  replace(is.na(.), 0)

train <- train_full %>% bind_cols(train_zero)

# Set a seed and create cross-validation folds
set.seed(101)
train_folds <- vfold_cv(train, v = 10, strata = action_taken)
```

```
35
36 # Create a boosting tree model
37 bt_model <-
38   boost_tree(
39     mtry = 40,
40     trees = 100,
41     min_n = 10,
42     tree_depth = 10,
43     learn_rate = 0.2,
44     loss_reduction = 0.01,
45     sample_size = 1,
46     stop_iter = 5
47   ) %>%
48   set_mode("classification") %>%
49   set_engine("xgboost")
50
51 # Define a recipe for data preprocessing
52 recipe <-
53   recipe(action_taken ~ ., data = train %>% select_if(~!is.
       character(.)) ) %>%
54   step_rm(id) %>%
55   step_impute_mean(all_numeric()) %>%
56   step_zv(all_predictors()) %>%
57   step_center(all_numeric()) %>%
58   step_dummy(all_nominal(), -all_outcomes())
59
60 # Create a workflow that includes the model and recipe
61 workflow <-
62   workflow() %>%
63   add_model(bt_model) %>%
64   add_recipe(recipe)
65
66 # Fit the model using cross-validation
67 crossval_fit <-
68   workflow %>%
69   fit_resamples(resamples = train_folds)
70
71 # Collect and display model evaluation metrics
72 crossval_fit %>% collect_metrics()
73
```

```r
# Fit the final model using the entire training dataset
final_fit <-
  workflow %>%
  fit(data = train)

# Read the test data from "test2.csv" and handle missing values in a similar way as training data
test <- read.csv("test2.csv")
na_cols <- names(test[, colSums(is.na(test)) > 0])
na_only <- names(test[, colMeans(is.na(test)) == 1.0])
test[na_cols] <- test[na_cols] %>% replace_na(list(
  ethnicity_of_applicant_or_borrower_1 = 4,
  ethnicity_of_applicant_or_borrower_2 = 4
  # ... (other columns)
))
names(test[, colSums(is.na(test)) > 0])

# Select columns without missing values and replace missing values in the other columns with zeros
train_full <- test %>% select(!na_only)
train_zero <- test %>%
  select(na_only) %>%
  replace(is.na(.), 0)

test <- train_full %>% bind_cols(train_zero)

# Make predictions using the final model
predictions <-
  final_fit %>%
  predict(new_data = test)

# Combine predictions with the "id" column and write to a CSV file
predictions <-
  test %>% select(id) %>%
  bind_cols(predictions)

write_csv(predictions, "predictions.csv")
```

8 R Code for Stacked Ensemble Model

```r
library(tidymodels)
library(tidyverse)
library(readr)
library(dplyr)
library(keras)
library(caret)
library(baguette)
library(parsnip)
library(doParallel)
library(stacks)
library(kernlab)

setwd("/Users/yasha/Downloads/classif_proj")
train2 <- read.csv("train2.csv")
train <- train2

train$action_taken <- as.factor(train$action_taken)

na_cols <- train %>%
  mutate_all(.funs = is.na)  %>% # Checks all columns for NA values
  summarise_all(sum) %>% # Sums greater than 0 indicate column contains NA's
  pivot_longer(cols = colnames(train)) %>% # Column names to Row names
  filter(value > 0) # Extracts column names containing NA values

empty_cols <- train %>%
  select(all_of(na_cols %>% pull(name))) %>% # Select all of the columns
  mutate_all(.funs = is.na)  %>%             # with NA values
  summarise_all(sum) %>%
  pivot_longer(cols = all_of(na_cols %>% pull(name))) %>%
  filter(value == nrow(train)) %>% # Filter columns that have only NA values
  pull(name) # Extract names of empty columns
```

```r


na_cols <- na_cols %>%
  filter(value != nrow(train) & name != "state") %>%
  pull(name)

train <- train %>%
  select(-all_of(empty_cols)) %>%  # Remove empty columns
  drop_na(state) # Remove rows where there is NA listed as
      the state

# Replaces columns with respective NA values from metadata.
    csv
train <- train %>%
  replace_na(list(ethnicity_of_applicant_or_borrower_1 = 4,
                  ethnicity_of_applicant_or_borrower_2 = 4,
                  ethnicity_of_applicant_or_borrower_3 = 4,
                  ethnicity_of_applicant_or_borrower_4 = 4,
                     # create new "NA" vals
                  ethnicity_of_applicant_or_borrower_5 = 4,
                  ethnicity_of_co_applicant_or_co_borrower_1
                       = 4,
                  ethnicity_of_co_applicant_or_co_borrower_2
                       = 4,
                  ethnicity_of_co_applicant_or_co_borrower_3
                       = 4,
                  ethnicity_of_co_applicant_or_co_borrower_4
                       = 4,
                  ethnicity_of_co_applicant_or_co_borrower_5
                       = 4,
                  race_of_applicant_or_borrower_1 = 7,
                  race_of_applicant_or_borrower_2 = 7,
                  race_of_applicant_or_borrower_3 = 7,
                  race_of_applicant_or_borrower_4 = 7,
                  race_of_applicant_or_borrower_5 = 7,
                  race_of_co_applicant_or_co_borrower_1 = 7,
                  race_of_co_applicant_or_co_borrower_2 = 7,
                  race_of_co_applicant_or_co_borrower_3 = 7,
                  race_of_co_applicant_or_co_borrower_4 = 7,
                  race_of_co_applicant_or_co_borrower_5 = 7,
```

```
                    age_of_applicant_62 = "8888",
                    age_of_co_applicant_62 = "8888",
                    automated_underwriting_system_2 = 6,
                    automated_underwriting_system_3 = 6,
                    prepayment_penalty_term = 0 #since if
                        there is no prepayment penalty term,
                        then the duration is 0 months
  ))

#train <- train %>%
#  mutate(age_of_applicant_or_borrower = ifelse(age_of_
    applicant_or_borrower == "9999", "8888", age_of_applicant
    _or_borrower))

#was only 1 observation, so it gave warning while training
    saying new factor
rows_with_9999 <- train %>%
  filter(age_of_applicant_or_borrower == 9999)

train <- train %>%
  filter(age_of_applicant_or_borrower != 9999)

# this leaves the columns - "income"  "combined_loan_to_
    value_ratio"  "loan_term"  "introductory_rate_period"  "
    property_value"  "multifamily_affordable_units"

train <- train %>%
  select(-c("id", "activity_year", "legal_entity_identifier_
      lei"))

#-------------------------------------------------------

train_folds <- vfold_cv(train, v = 5) #10 takes a while

#Recipe

rec <- recipe(action_taken ~ ., data=train) %>%
  step_impute_mean(c("income", "combined_loan_to_value_ratio
      ", "loan_term", "introductory_rate_period", "property_
      value", "multifamily_affordable_units")) %>%   #knn
```

```
      taking so long
  step_zv(all_predictors()) %>%
  step_normalize(all_numeric()) %>%
  step_dummy(all_nominal(), -all_outcomes())

ctrl_res <- control_stack_resamples()
#----------------------MLP---------------------------
mlp_model <- mlp(
  activation = "relu",          #no need for bag_mlp, can try
      elu also
  learn_rate = 0.01,
  dropout = 0.2,
  hidden_units = 40,
  penalty = 0,
  epochs = 50
) %>%
  set_engine("keras") %>%       #keras -- mlp, nnet -- bag_
     mlp
  set_mode("classification")

mlp_workflow <- workflow() %>%
  add_recipe(rec) %>%
  add_model(mlp_model)

mlp_results <- fit_resamples(
  mlp_workflow,
  resamples = train_folds,
  metrics = metric_set(roc_auc, accuracy),
  control = ctrl_res
)
mlp_results %>% collect_metrics()

#-----------------------BOOSTTREE
    --------------------------------
bt_model <- boost_tree(
  mtry = 40,   #for rand forest also
  trees = 60,   #for rand forest also
  min_n = 10,   #for rand forest also
  tree_depth = 10,
  learn_rate = 0.23,
```

```
  loss_reduction = 0.01,
  sample_size = 1,
  stop_iter = 5
) %>%
  set_engine("xgboost") %>%       #keras -- mlp, nnet -- bag
      _mlp
  set_mode("classification")

bt_workflow <- workflow() %>%
  add_recipe(rec) %>%
  add_model(bt_model)

bt_results <- fit_resamples(
  bt_workflow,
  resamples = train_folds,
  metrics = metric_set(roc_auc, accuracy),
  control = ctrl_res
)
bt_results %>% collect_metrics()

#------------------------LOGISITCREG
    ----------------------------

lr_model <- logistic_reg(
  penalty = 0.5,
  mixture = 0.5
) %>%
  set_engine("glm") %>%       #keras -- mlp, nnet -- bag_mlp
  set_mode("classification")

lr_workflow <- workflow() %>%
  add_recipe(rec) %>%
  add_model(lr_model)

lr_results <- fit_resamples(
  lr_workflow,
  resamples = train_folds,
 metrics = metric_set(roc_auc, accuracy),
  control = ctrl_res
)
```

```
lr_results %>% collect_metrics()

#----------------------------SVMLINEAR---------------------

# svm_model <- svm_linear(
#   cost = 1,
#   margin = 0.1,
# ) %>%
#   set_mode("classification") %>%
#   set_engine("kernlab")
#
# svm_workflow <- workflow() %>%
#   add_recipe(rec) %>%
#   add_model(svm_model)
#
# svm_results <- fit_resamples(
#   svm_workflow,
#   resamples = train_folds,
#   metrics = metric_set(roc_auc, accuracy),
#   control = ctrl_res
# )
# svm_results %>% collect_metrics()

#-------------------------------------------------
#0.982, 85,85
#Fit the workflow to the entire training data
final_model <- stacks()
final_model <-
  stacks() %>%
  add_candidates(mlp_results) %>%
  add_candidates(bt_results) %>%
add_candidates(lr_results) #%>%
  #add_candidates(svm_results)

test <- read.csv("test2.csv")
na_cols <- test %>%
  mutate_all(.funs = is.na)  %>% # Checks all columns for NA
      values
  summarise_all(sum) %>% # Sums greater than 0 indicate
     column contains NA's
```

```
206   pivot_longer(cols = colnames(test)) %>% # Column names to
          Row names
207   filter(value > 0) # Extracts column names containing NA
          values
208
209 empty_cols <- test %>%
210   select(all_of(na_cols %>% pull(name))) %>% # Select all of
          the columns
211   mutate_all(.funs = is.na)  %>%              # with NA
          values
212   summarise_all(sum) %>%
213   pivot_longer(cols = all_of(na_cols %>% pull(name))) %>%
214   filter(value == nrow(test)) %>% # Filter columns that have
          only NA values
215   pull(name) # Extract names of empty columns
216
217
218 na_cols <- na_cols %>%
219   filter(value != nrow(test) & name != "state") %>%
220   pull(name)
221
222 test <- test %>%
223   select(-all_of(empty_cols))  # Remove empty columns
224 #%>% drop_na(state) # Remove rows where there is NA listed
      as the state
225 # Replaces columns with respective NA values from metadata.
      csv
226 test <- test %>%
227   replace_na(list(ethnicity_of_applicant_or_borrower_1 = 4,
228                   ethnicity_of_applicant_or_borrower_2 = 4,
229                   ethnicity_of_applicant_or_borrower_3 = 4,
230                   ethnicity_of_applicant_or_borrower_4 = 4,
                        # create new "NA" vals
231                   ethnicity_of_applicant_or_borrower_5 = 4,
232                   ethnicity_of_co_applicant_or_co_borrower_1
                        = 4,
233                   ethnicity_of_co_applicant_or_co_borrower_2
                        = 4,
234                   ethnicity_of_co_applicant_or_co_borrower_3
                        = 4,
```

```
                    ethnicity_of_co_applicant_or_co_borrower_4 = 4,
                    ethnicity_of_co_applicant_or_co_borrower_5 = 4,
                    race_of_applicant_or_borrower_1 = 7,
                    race_of_applicant_or_borrower_2 = 7,
                    race_of_applicant_or_borrower_3 = 7,
                    race_of_applicant_or_borrower_4 = 7,
                    race_of_applicant_or_borrower_5 = 7,
                    race_of_co_applicant_or_co_borrower_1 = 7,
                    race_of_co_applicant_or_co_borrower_2 = 7,
                    race_of_co_applicant_or_co_borrower_3 = 7,
                    race_of_co_applicant_or_co_borrower_4 = 7,
                    race_of_co_applicant_or_co_borrower_5 = 7,
                    age_of_applicant_62 = "8888",
                    age_of_co_applicant_62 = "8888",
                    automated_underwriting_system_2 = 6,
                    automated_underwriting_system_3 = 6,
                    prepayment_penalty_term = 0 #since if there is no prepayment penalty term, then the duration is 0 months
  ))

test <- test %>%
  select(-c("activity_year", "legal_entity_identifier_lei"))

final_model <- final_model %>%
  blend_predictions()

autoplot(final_model)
autoplot(final_model, type = "weights")

final_model <- final_model %>%
  fit_members()

collect_parameters(final_model)

predictions <- final_model %>%
  predict(test) %>%
  cbind(test %>% select(id))
```

```
271
272 nrow(predictions)      #need 117593 rows
273
274 write_csv(x = predictions %>% rename(action_taken = .pred_
        class),
275           file = "predictions.csv")
```