STATS 101C Summer 2023

Kaggle Regression Competition
Team 12 Report

Introduction: Context and Background Information

Variables associated with response based on background info

Race plays a pivotal role in shaping voters' choices, often driven by societal and historical factors. For instance, African Americans tend to favor the Democratic Party due to its historical commitment to civil rights, while a majority of White voters support the Republican Party. Asian Americans also lean Democratic to some extent. Education is another influential factor, with those holding college degrees, including post-graduates, being more inclined toward the Democratic Party. Income per capita can serve as an indicator of wealth, with greater wealth associated with Democratic voters. Age plays a role as well, as older voters are generally more likely to vote for Republicans, while younger demographics often lean Democratic. These factors, intertwined with political dynamics, collectively shape the diverse landscape of voter responses in U.S. elections.

Citations

- https://www.pewresearch.org/politics/2020/08/13/important-issues-in-the-2020-election/
- https://www.pewresearch.org/politics/2023/07/12/demographic-profiles-of-republican-and-democratic-voters/
- https://recipes.tidymodels.org/reference/step_impute_bag.html
- https://www.tidymodels.org/find/parsnip/
- https://stacks.tidymodels.org/articles/basics.html
- https://recipes.tidymodels.org/reference/step_YeoJohnson.html

Exploratory Data Analysis

Interactions between variables

Since there are many overlapping predictors, including all of them might make our model way too complicated and further cause an overfitting problem. Thus, we decide to remove variables that are completely overlapped by others and not provide new information. The following plots are created based on these selected variables. We further combined some of these variables in the preprocessing step.

We created a table for correlation between selected variables and set threshold $= 0.8$. There exist columns names that are duplicated (ie. both x0035e and x0058e are named “Total Population : Two or more races”), and there also exist columns partially overlapping (ie. total population of 25 years old and overlap with population of females), so it makes sense to have many predictors with high correlation.

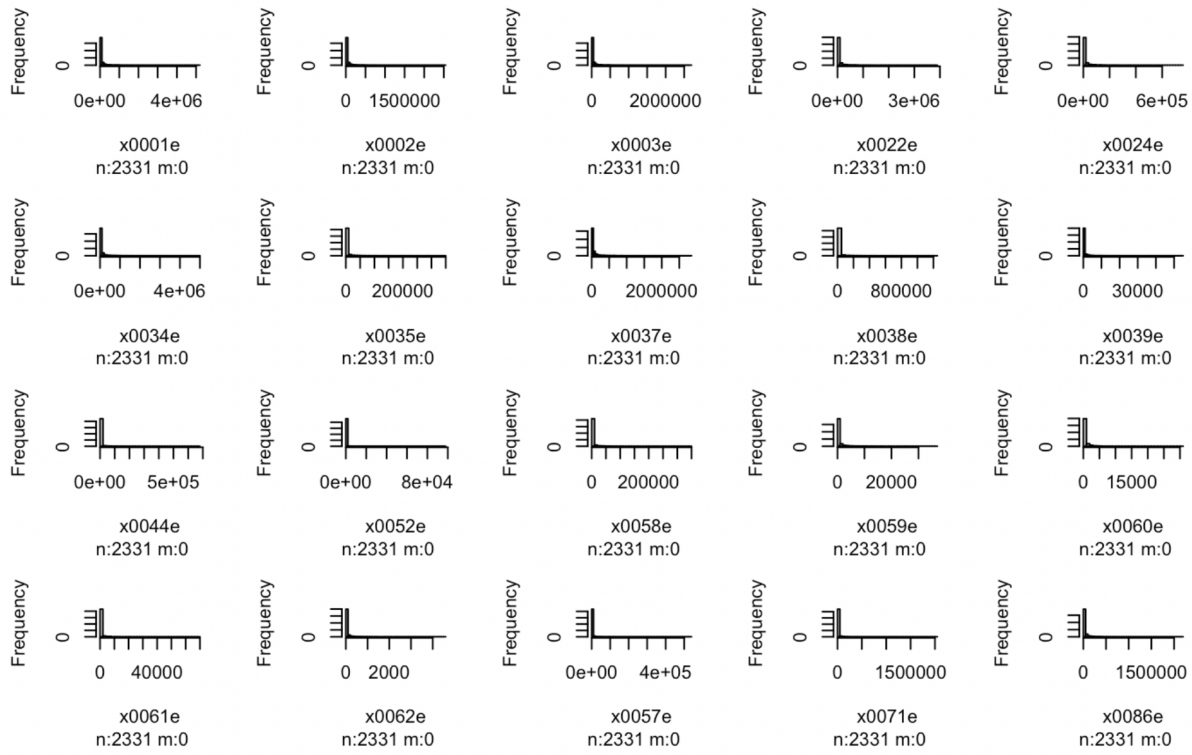
	Var1 <fctr>	Var2 <fctr>	Freq <dbl>
499	x0035e	x0058e	1.0000000
824	x0022e	c01_006e	0.9999106
83	x0001e	x0003e	0.9998523
42	x0001e	x0002e	0.9998410
208	x0003e	x0034e	0.9996981
206	x0001e	x0034e	0.9996423
84	x0002e	x0003e	0.9993867
207	x0002e	x0034e	0.9992722
126	x0003e	x0022e	0.9991171
124	x0001e	x0022e	0.9990153

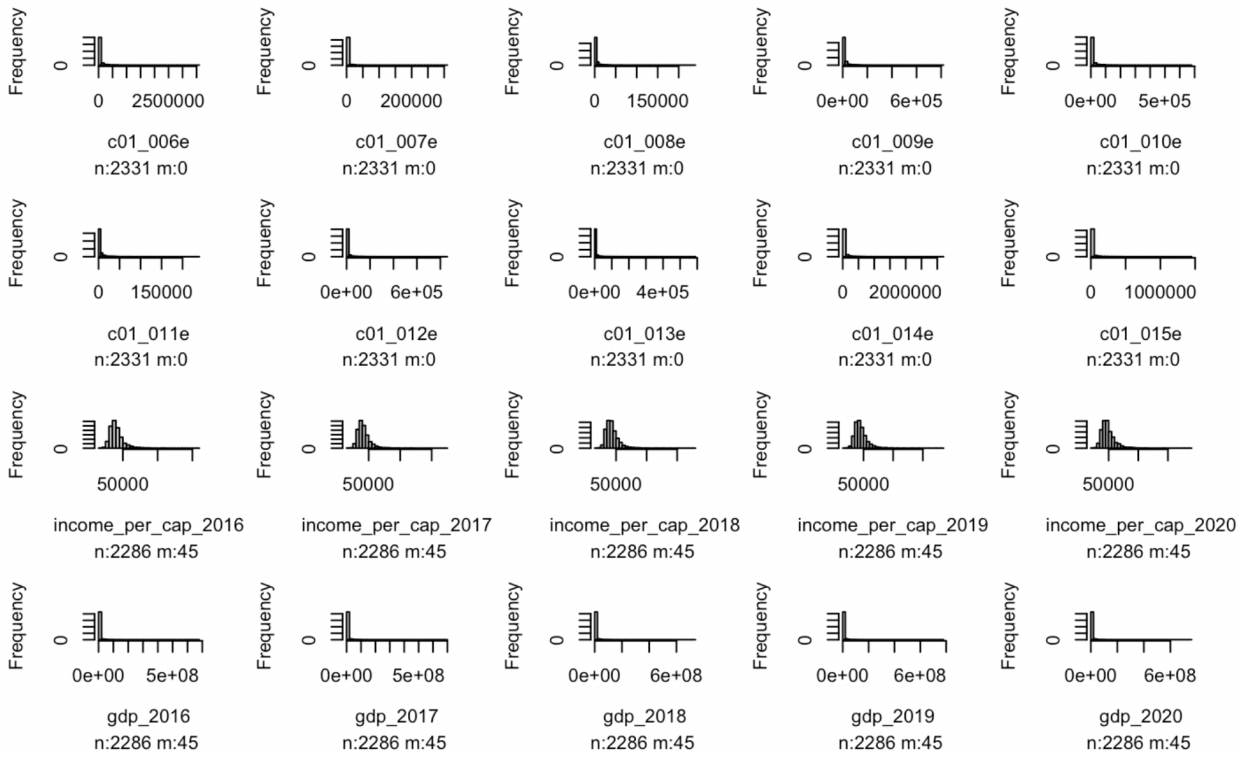
1–10 of 280 rows Previous 1 2 3 4 5 6 … 28 Next

To better understand the predictors affecting the target variable. We decided to look into the relationships between some of the potential predictors and the target variable. For example, the geographical aspect, the areas of urban or rural play an important role in shaping voting behavior. Predictors related to income per capita data is also important as those data shows how well the economy is and this may greatly influence voters’s perceptions of their government policies. We also want to look into racial distributions and educational backgrounds, etc, since we believe that those predictors might also have significant associations with our target variable ‘percent_dem’.

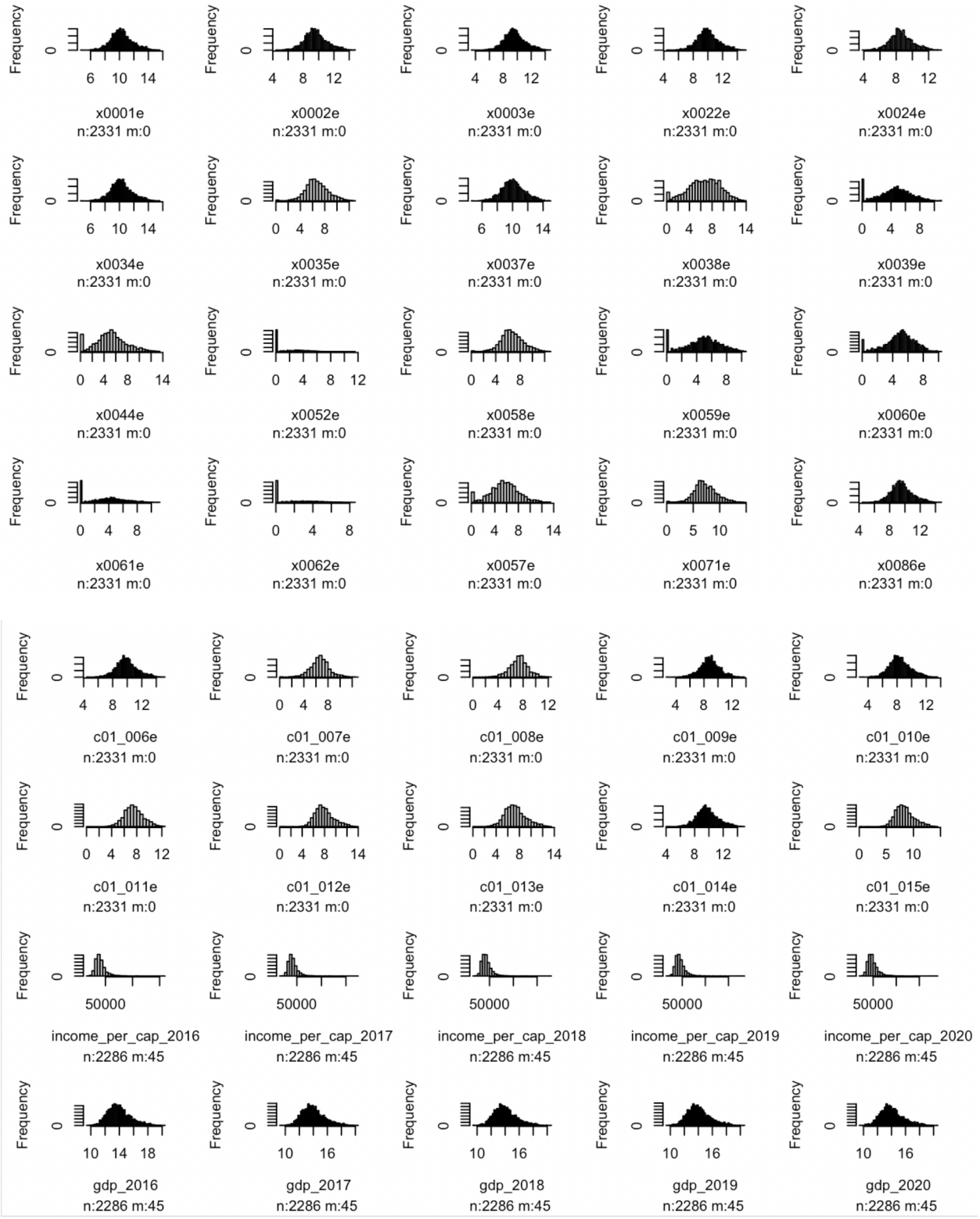
Visualizations

1. The histograms of all kept predictors reveal a common pattern of significant right-skewness. This skewness implies that counties with a lower prevalence of a particular category of voters tend to have a higher likelihood of voting for the Democratic party. However, there are notable exceptions in the case of predictors related to income per capita from 2016 to 2020, which exhibit a normal distribution with a symmetric shape. The prevalent right-skewness across most predictors suggests the potential need for transformations.



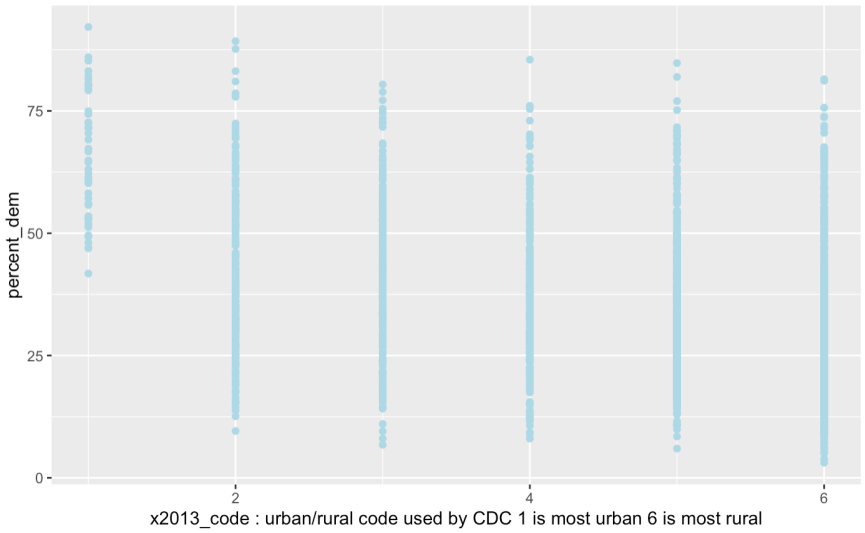


2. Following the application of log transformation to all retained predictors, with the exception of income per capita from 2016 to 2020, a noticeable and statistically significant change is observed in the distribution of most of these transformed predictors. The transformation has led to a more symmetrical and approximately normal distribution for many of these variables. Consequently, log transformation emerges as a viable consideration for incorporation into our modeling process.

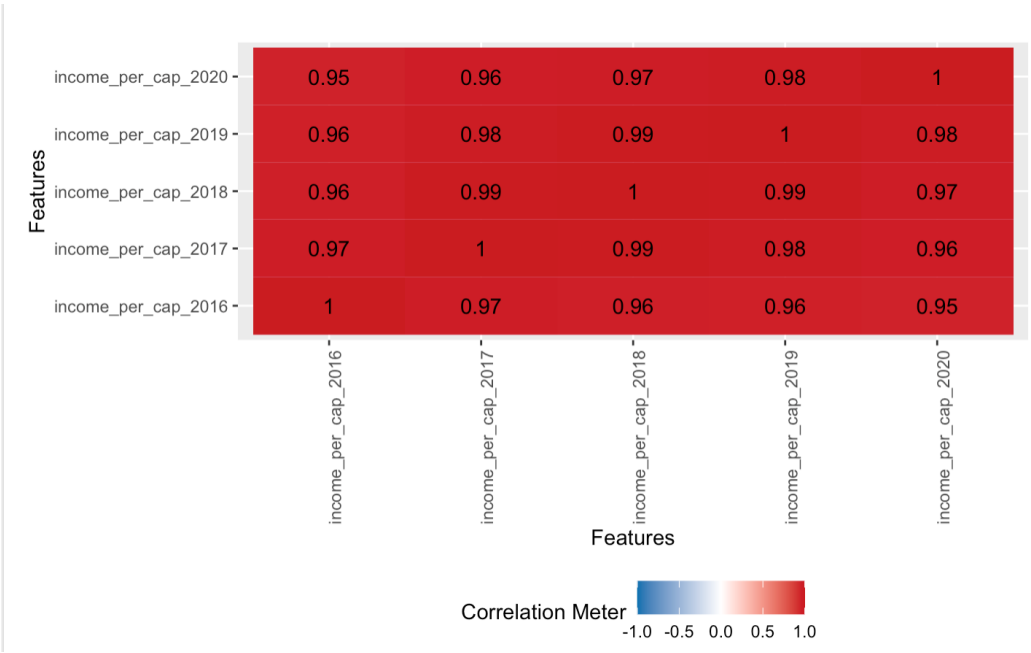


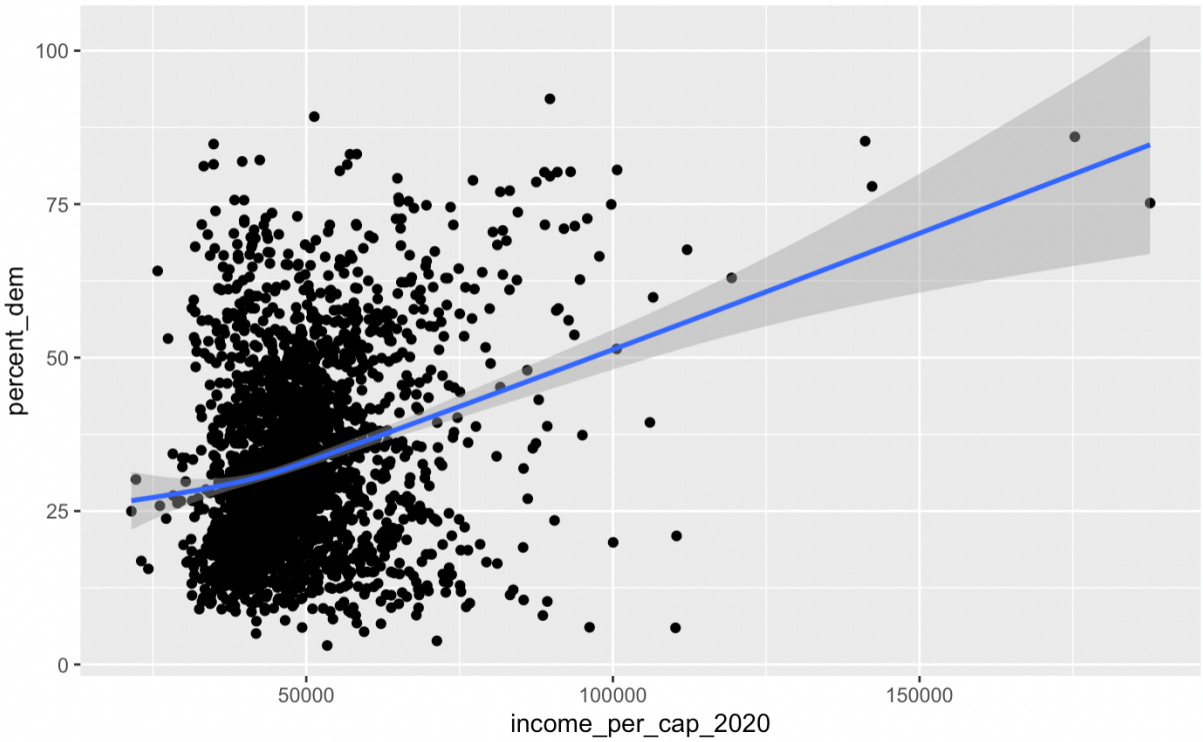
3. Despite being encoded as numerical values, 'x2013_code' is a categorical predictor representing the degree of rurality in counties. The plot illustrates that in counties

categorized as '1' (the most urban areas), there is a preference for the Democratic party, with a percentage of votes ranging from 40% to 90%. In contrast, counties classified as '6' (the most rural areas) exhibit a broader range of 'percent_dem' values, indicating a more diverse voting pattern.



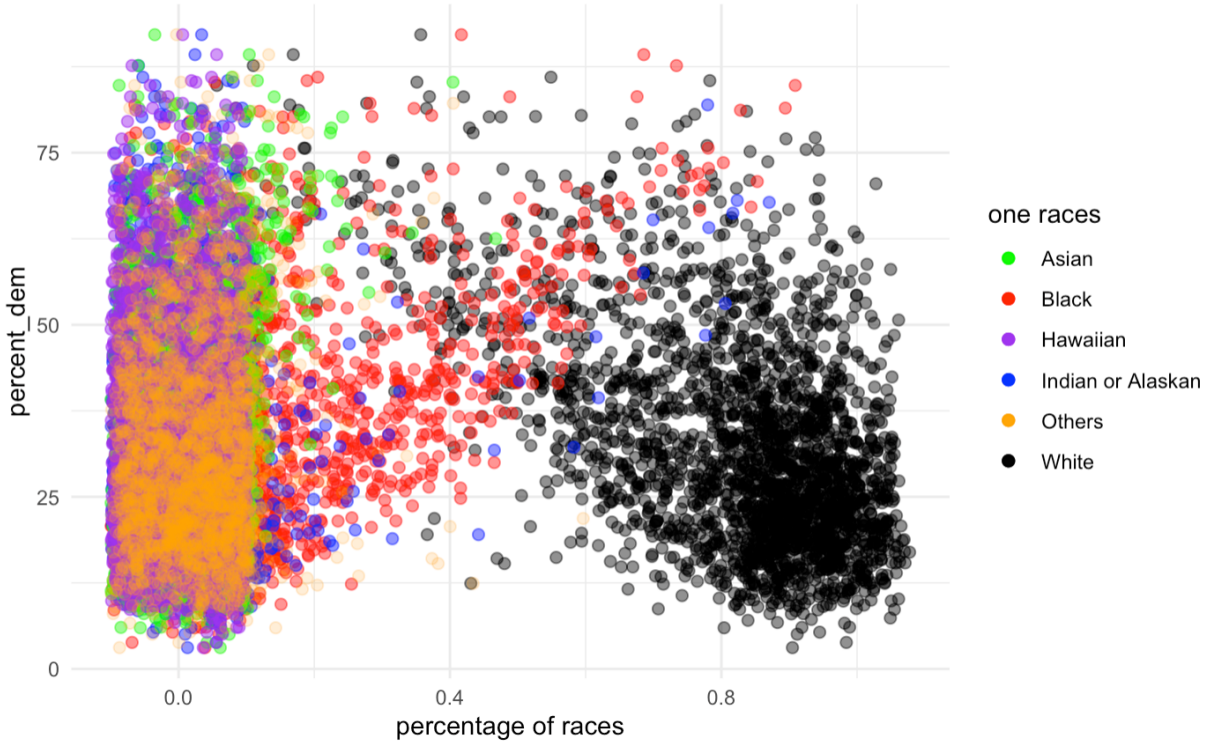
4. We observe consistent trends in the plots of 'income_per_cap' vs. 'percent_dem' from 2016 to 2020. Specifically, as the income per capita of a county increases, there is a corresponding increase in the percentage of votes for the Democratic party ('percent_dem'). The correlation between income per capita across these years appears logical, as counties with higher income levels tend to maintain their relative economic standing over this time period.



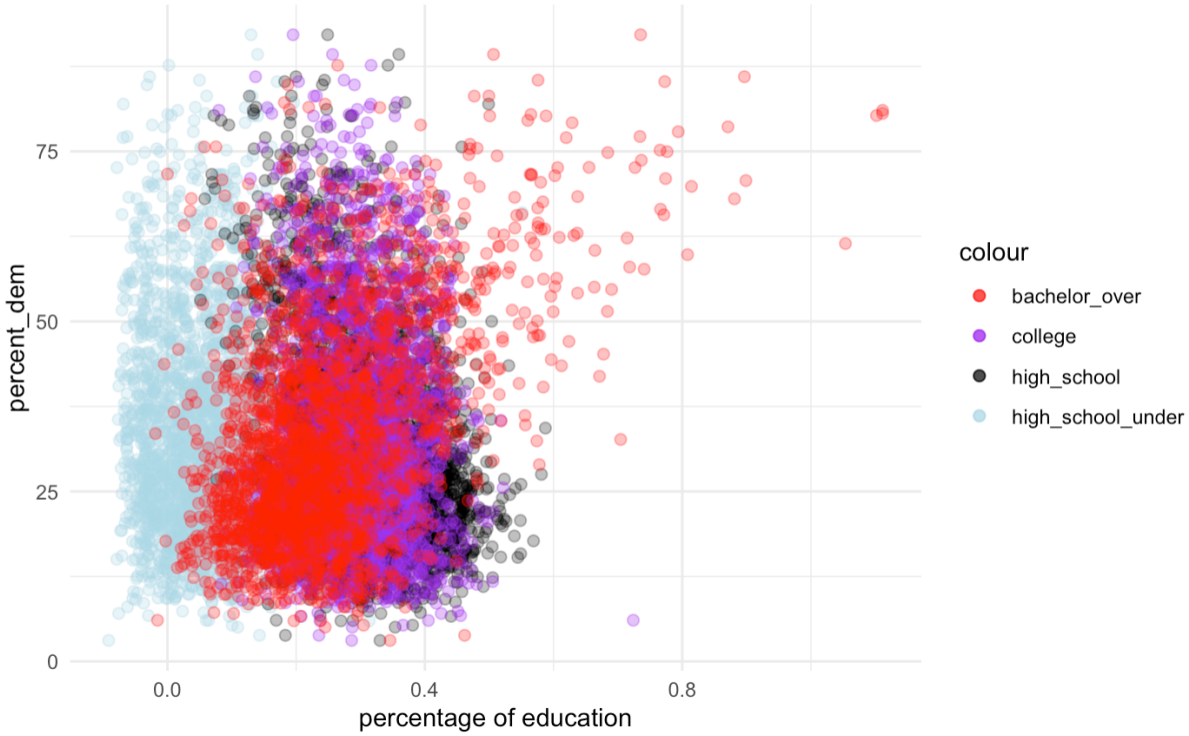


5. We calculated the percentage of voters of a specific race within each county by dividing the population of that race by the total county population. Notably, our analysis reveals a significant trend: as the percentage of minority voters (those other than White) increases within counties, there is a corresponding increase in the percentage of people who vote

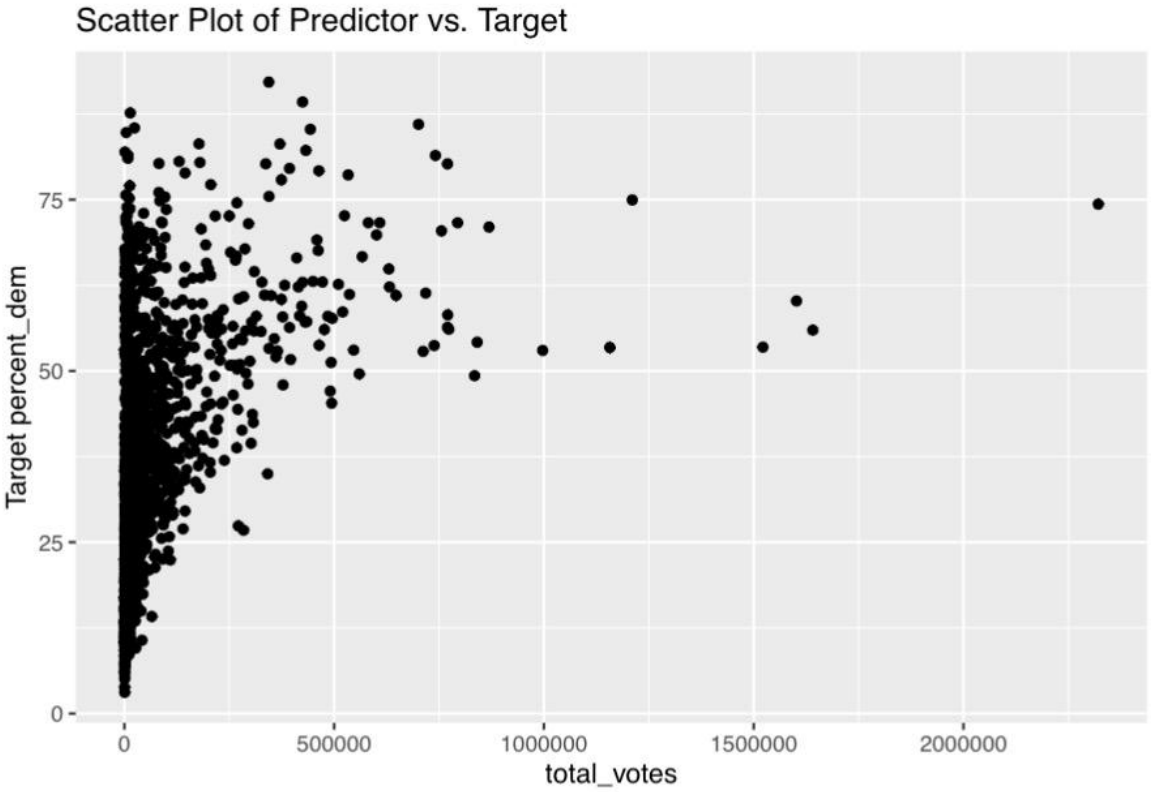
for the Democratic party. However, it's interesting to observe that even in counties with a high percentage of both White residents and Democratic voters, counties with the highest percentage of White residents tend to have a lower proportion of Democratic voters. This suggests a complex relationship between racial demographics and political preferences.

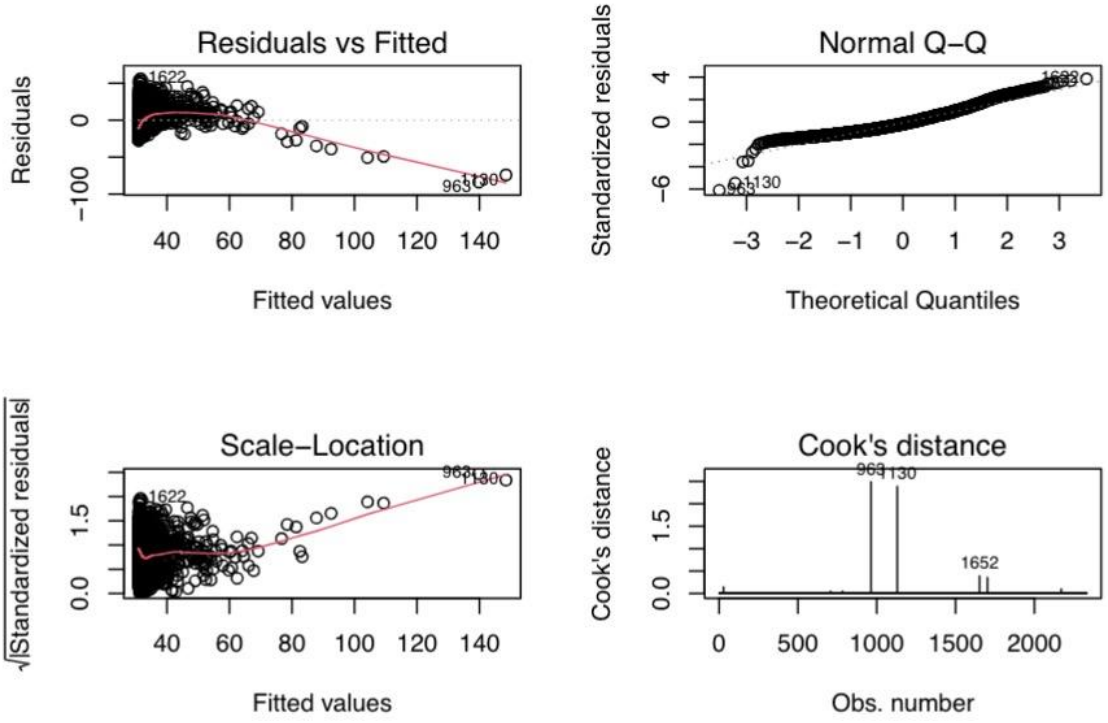


6. The percentage of education for each county was calculated by considering the population of voters with an education level above a specified threshold relative to the total population, including both voters below 24 years of age and those aged 25 and above. The plot reveals that there is no significant difference between the level of education and voter preferences. Interestingly, counties with a higher percentage of voters holding bachelor's degrees or higher tend to exhibit lower chances of voting for the Democratic party.



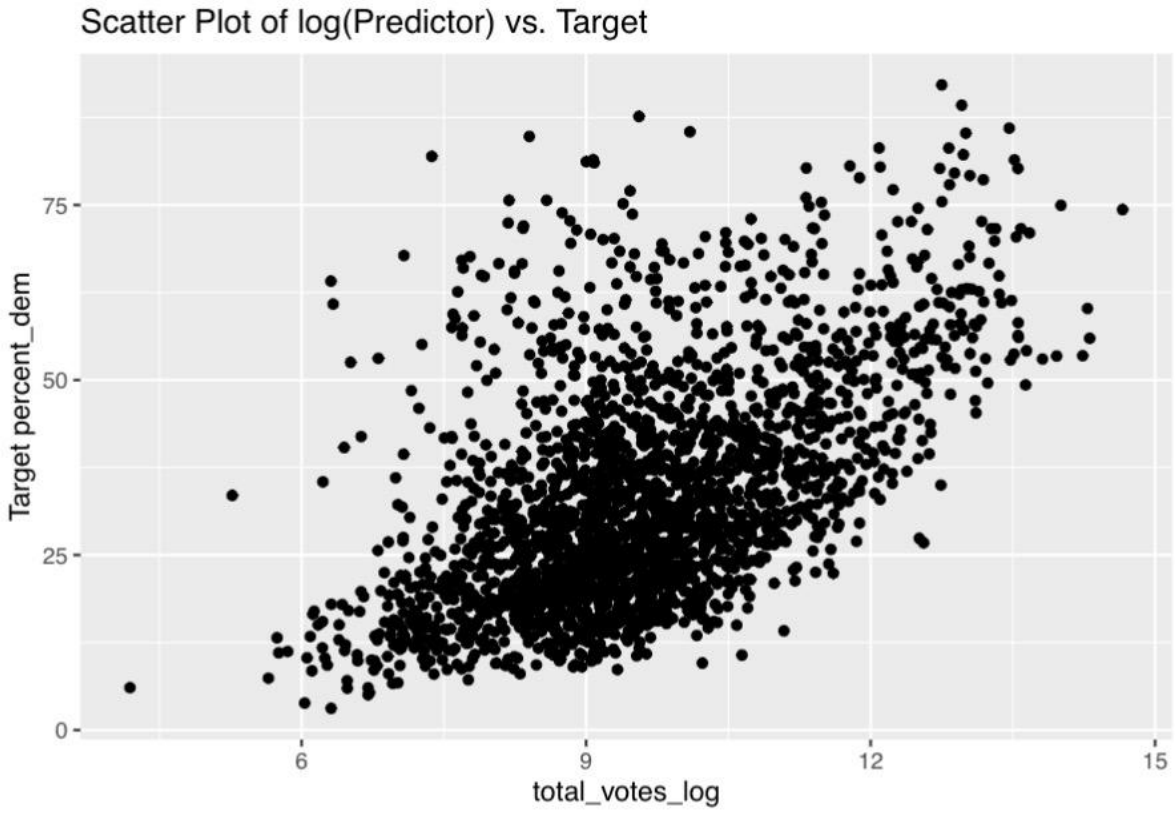
7. We tried to look at the linear relationship between the target variable and some of our significant predictor variables by using scatter plot and linear model, and found out transformations such as log-transformation are needed for many of our predictors.(‘total_votes’, ‘x0001e’, etc.) For example, ‘total_votes’, the Total Votes Cast. We could not determine the relationship between the predictor and the target variable based on the scatter plot before any transformation. And the data points on the generated Residual vs Fitted plots are clustered instead of randomly scattered, which tells us that some kind of data transformation is needed for a better fit.

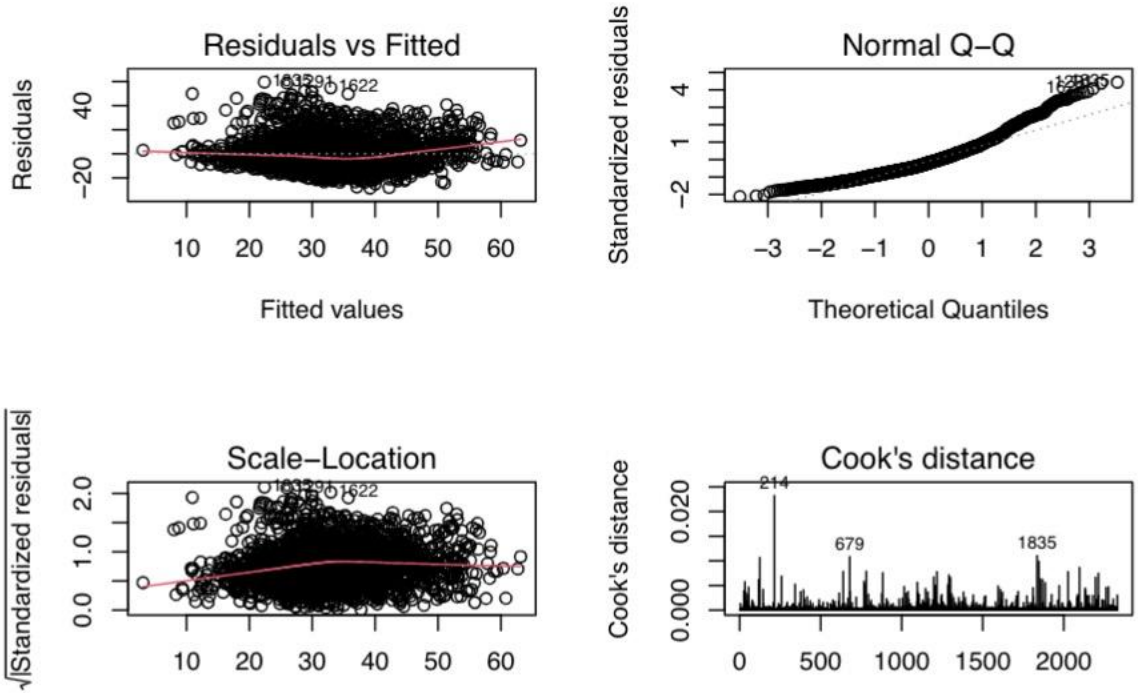




8. We decided to do a log-transformation on predictor ‘total_votes’ and got much better results. From the scatter plot, we can see a clear positive correlation between the

predictor variables(‘total_votes’) and the target variable(‘percent_dem’), which helped us to determine the relationship between them. The new Residual vs Fitted plot has randomly scattered data points that tells us whether a linear regression model is the correct model to pick. And the uniform spread of points above and below zero along the horizontal axis also tells us that the variance of the residuals is constant across all levels of the predictor variables. The plot also helped us identify some of the potential outliers.





At the end, we decided to skip those predictors who are duplicated or overlapping with others and selected those predictors with higher correlation to the target variable for further processing and analysis. We also planned to add some new columns with adjusted values of some predictors.

Preprocessing / Recipes

Firstly, we create the basic recipe using our selected columns.We choose income per capita and GDP for observing the micro and macro economic effects on the percentage of voting. In **`step_mutate`**, we create a new column with race, age, and gender divided by the total population because we want to compare counties under the same scales. We also created new columns with different education levels. Since the percentage of social identity columns is already counted, we use the step remove function to remove the columns and id in case these unscaled columns interfere with the prediction.

`Step_naomit` removes the counties where there is no observation voting for Biden in the training data. **`Step_impute_bag`** replaced the missing value with the reasonable value from bagged tree models which also allowed the potential of having different types of data. We

use **step_nzv to** remove the column with no variance since those predictors might not be greatly involved in the predictions. We use **step_dummy** to transform the potential categorical variables into numeric variables to fit in the numeric prediction. **Step_normalize** normalizes all the predictors to make the data more flexible and compatible by deleting redundancy. Furthermore, it helps the data to meet the assumption if we have to use some specific model in future.

```
# The basic recipe of the models
basic_recipe <-recipe(percent_dem ~ id + x0001e + x0002e + x0003e +
x0022e + x0024e + x0034e + x0035e +  x0037e + x0038e + x0039e + x0044e
+ x0052e + x0058e + x0059e + x0060e + x0061e + x0062e + x0057e +
x0071e + x0086e + c01_006e + c01_007e + c01_008e + c01_009e + c01_010e
+ c01_011e + c01_012e + c01_013e + c01_014e + c01_015e +
income_per_cap_2016 + income_per_cap_2017 + income_per_cap_2018 +
income_per_cap_2019 + income_per_cap_2020 + gdp_2016 + gdp_2017 +
gdp_2018 + gdp_2019 + gdp_2020 + x2013_code, data = train) %>%
  # convert some columns as ratio and mutate them as new columns
  step_mutate(
    male_perc = x0002e / x0001e,
    female_perc = x0003e / x0001e,
    adult_pop = x0022e - x0024e,
    elder_pop = x0024e,
    white_race_perc = x0037e / x0001e,
    black_race_perc = x0038e / x0001e,
    american_race_perc = x0039e / x0001e,
    asian_race_perc = x0044e / x0001e,
    hawaiian_race_perc = x0052e / x0001e,
    other_race_perc = x0057e / x0001e,
    hisp_race_perc = x0071e / x0001e,
    mix_wb_perc = x0059e / x0001e,
    mix_wam_perc = x0060e / x0001e,
    mix_wai_perc = x0061e / x0001e,
    mix_bam_perc = x0062e / x0001e,
    high_school = c01_014e - c01_015e,
    bachelor_over = c01_015e,
    high_school_under = c01_006e - c01_014e) %>%
  # remove the original columns and the id column
  step_rm(
    x0001e, x0002e, x0003e, x0024e, x0022e, x0034e, x0035e, x0037e ,
x0038e , x0039e , x0044e , x0052e, x0057e, x0071e , x0059e , x0058e,
x0060e , x0061e , x0062e , c01_014e , c01_006e, c01_015e) %>%
  step_rm(id) %>%
  # remove rows with missing values in percent_dem
```

```
  # fill in missing values using bagged tree models for predictors
  step_naomit(percent_dem) %>%
  step_impute_bag(all_predictors()) %>%
  # remove predictors with low variance
  step_nzv(all_predictors()) %>%
  # create dummy variables for categorical variables
  step_dummy(all_nominal()) %>%
  # standardize numeric variables
  step_normalize(all_numeric_predictors())
```

After having the basic recipe, we structure various recipes to fit in a different model. The first recipe is the spline recipe which is structured for the models that require the strictest assumptions. **Step_corr** removes predictors that are highly correlated. **Step_lincomb** removes the redundant variables which are the combination of other variables in case we miss to remove from the model selection and basic recipe. **Step_bs** adds new columns with new features that enable modeling variables in a nonlinear manner. As we already normalized all the predictors, we use the YeoJohnson transformation, **step_YeoJohnson**, instead of log transformation to transform predictors including all the possible negative values that come from the normalization.

```
# recipe for spline model
spline_rec <- basic_recipe %>%
  #removing high correlated variables
  step_corr(all_numeric(), -all_outcomes()) %>%
  # creating linear combination of numeric features
  step_lincomb(all_numeric(), -all_outcomes()) %>%
  # removing categorical variables
  step_rm(all_nominal()) %>%
  # Yeo Johnson transformation on predictors
  step_bs(all_predictors()) %>%
  step_YeoJohnson(all_predictors())
```

The second recipe is for the model which is less strict. We only focus on removing the high-correlated variables with the **Step_corr** function.

```
# recipe for svm model
svm_rec <- basic_recipe %>%
  step_corr(all_predictors())
```

The third recipe is preparing for capturing the info to do PCA model analysis. **Step_lincomb** removes the redundant variables from the model selection and basic recipe.

Step_other collapses nominal variables whose individual proportions are small. Although we already transform our nominal variables into dummy variables in the basic recipe. We add this step to confirm that the preparation of the model is fully made. After we standardize and normalize through the above steps, **Step_pca** helps produce the info that might be significant but hasn't been mutated and specified in the basic recipe for following model analysis. Setting tune variables in num_comp allows us to tune how many new predictors are retained will give us the best result.

```
# recipe for pca model
pca_rec_tuning <- basic_recipe %>%
  step_lincomb(all_numeric(), -all_outcomes()) %>%
  step_other(all_nominal()) %>%
  step_pca(all_predictors(), num_comp = tune())
```

Candidate Models / Model Evaluation / Tuning

Our group experimented with several models, such as linear regression (with PCA and spline), random forest, boosted trees, k-nearest neighbors (KNN), and support vector machines (SVM). We first decided to tune our models.

- PCA: We aim to optimize the hyperparameter `num_comp` in `step_pca(all_predictors(), num_comp = tune())` through a random search grid with a size of 10.
- Random Forest: We aim to optimize the hyperparameter `min_n` and `trees` in `rand_forest(min_n = tune(), trees = tune())` through a grid search with a level of 3.
- Boosted Trees: We aim to optimize the hyperparameter `learn_rate, trees,` and `tree_depth` in `boost_tree(learn_rate = tune(), trees = tune(), tree_depth = tune())` through a grid search with a level of 3.
- KNN: We aim to optimize the hyperparameter `neighbors` in `nearest_neighbor(neighbors = tune("k"))` through a grid search with a level of 6.
- SVM: We aim to optimize the hyperparameter `rbf_sigma` and `cost` in `svm_rbf( cost = tune("cost"), rbf_sigma = tune("sigma"))` through a grid

search with a level of 6 and consider values of cost that is less than or equal to 0.05 (`filter = c(cost <= 0.05)`).

For evaluation, we primarily examine the value of RMSE. We employ the "`show_best()`" function to examine the parameters that lead to a decrease in RMSE. The following are the outcomes.

PCA:

tibble: 4 × 7

num_comp <int>	.metric <chr>	.estimator <chr>	mean <dbl>	n <int>	std_err <dbl>	.config <chr>
4	rmse	standard	11.80477	10	0.2073285	Preprocessor1_Model1
3	rmse	standard	12.06759	10	0.1586751	Preprocessor3_Model1
1	rmse	standard	14.56385	10	0.1690162	Preprocessor4_Model1
2	rmse	standard	14.58031	10	0.1796279	Preprocessor2_Model1

Spline:

1 × 6

:	.estimator <chr>	mean <dbl>	n <int>	std_err <dbl>	.config <chr>
	standard	14.85359	10	3.026755	Preprocessor1_Model1

Random Forest

tibble: 5 × 8

trees <int>	min_n <int>	.metric <chr>	.estimator <chr>	mean <dbl>	n <int>	std_err <dbl>	.config <chr>
1000	2	rmse	standard	8.490421	10	0.1308223	Preprocessor1_Model2
2000	2	rmse	standard	8.499208	10	0.1322694	Preprocessor1_Model3
2000	21	rmse	standard	8.664706	10	0.1354097	Preprocessor1_Model6
1000	21	rmse	standard	8.667835	10	0.1398883	Preprocessor1_Model5
1000	40	rmse	standard	8.862086	10	0.1483633	Preprocessor1_Model8

Boost Tree

A tibble: 5 × 9

trees <int>	tree_depth <int>	learn_rate <dbl>	.metric <chr>	.estimator <chr>	mean <dbl>	n <int>	std_err <dbl>	.config <chr>
2000	8	0.01778279	rmse	standard	7.897124	10	0.10262061	Preprocessor1_Model15
1000	8	0.01778279	rmse	standard	7.902655	10	0.10297443	Preprocessor1_Model14
2000	1	0.31622777	rmse	standard	8.255560	10	0.06815848	Preprocessor1_Model09
1000	1	0.31622777	rmse	standard	8.279749	10	0.08259296	Preprocessor1_Model08
2000	15	0.01778279	rmse	standard	8.280869	10	0.12985853	Preprocessor1_Model24

SVM:

A tibble: 5 × 8

cost <dbl>	sigma <dbl>	.metric <chr>	.estimator <chr>	mean <dbl>	n <int>	std_err <dbl>	.config <chr>
0.0078125000	1e-02	rmse	standard	14.12467	10	0.1762308	Preprocessor1_Model10
0.0078125000	1e+00	rmse	standard	16.03918	10	0.2356276	Preprocessor1_Model12
0.0009765625	1e-02	rmse	standard	16.13939	10	0.2202588	Preprocessor1_Model09
0.0009765625	1e+00	rmse	standard	16.42264	10	0.2296994	Preprocessor1_Model11
0.0078125000	1e-04	rmse	standard	16.45675	10	0.2271409	Preprocessor1_Model08

KNN:

A tibble: 5 × 7

k <int>	.metric <chr>	.estimator <chr>	mean <dbl>	n <int>	std_err <dbl>	.config <chr>
15	rmse	standard	9.783966	10	0.1608243	Preprocessor1_Model6
12	rmse	standard	9.838005	10	0.1708951	Preprocessor1_Model5
9	rmse	standard	9.969306	10	0.1872222	Preprocessor1_Model4
6	rmse	standard	10.262562	10	0.2145962	Preprocessor1_Model3
3	rmse	standard	11.146245	10	0.2547419	Preprocessor1_Model2

By comparing the RMSE values, we recognize that the **random forest** and **boost tree** have outstanding performance. After tuning the parameters and comparing RMSE values, these are the final eight models we have selected as our candidates.

1. A linear regression with PCA. After tuning the parameter `'num_comp'`, we got the suggestion of `num_comp = 4`. It returns a RMSE of 11.80477.
2. A linear regression with Spline. This model experiences a `YeoJohnson` transformation. It does not have a hyperparameter for tuning. Hence the RMSE is 14.85359.
3. A random forest model. After tuning the parameters `'trees'` and `'min_n'`, we got the suggestion of `'trees = 1000'` and `'min_n = 2'`. It returns a RMSE of 8.490421.
4. A random forest model. After tuning the parameters `'trees'` and `'min_n'`, we got the suggestion of `'trees = 2000'` and `'min_n = 2'`. It returns a RMSE of 8.499208.
5. A boost tree model. After tuning the parameters `'trees', 'tree_depth'` and `learn_rate`, we got the suggestion of `'trees = 2000', 'tree_depth = 8'` and `'learn_rate = 0.01778279'`. It returns a RMSE of 7.897124.

6. A boost tree model. After tuning the parameters `'trees', 'tree_depth'` and `learn_rate`, we got the suggestion of `'trees = 1000', 'tree_depth = 8'` and `'learn_rate = 0.01778279'`. It returns a RMSE of 7.902655.
7. A SVM model. After tuning the parameters `'cost'` and `'rbf_sigma'`, we got the suggestion of `'cost = 0.0078125000'` and `'rbf_sigma = 1e-02'`. It returns a RMSE of 14.12467.
8. A KNN model. After tuning the parameters `'neighbors'`, we got the suggestion of `'neighbors = 15'`. It returns a RMSE of 9.783966.

Model Identifier	Type of Model	Engine	Recipe	Hyperparameters
linear_reg() step_pca(all_predictors(), num_comp = tune())	Linear regression	lm	pca_rec_tuning	num_comp = 4
linear_reg() step_YeoJohnson(all_predictors())	Linear regression	lm	spline_rec	/
rand_forest()	Random forest	ranger	basic_recipe	trees = 2000 min_n = 2
rand_forest()	Random forest	ranger	basic_recipe	trees = 1000 min_n = 2
boost_tree()	Boost tree	xgboost	basic_recipe	learn_rate = 0.01778279 trees = 8 tree_depth = 1000
boost_tree()	Boost tree	xgboost	basic_recipe	learn_rate = 0.01778279 trees = 8 tree_depth = 2000
svm_rbf()	svm	kernlab	svm__rec	rbf_sigma = 1e-02 cost = 0.0078125000
nearest_neighbor()	knn	kknn	basic_recipe	neighbors = 15

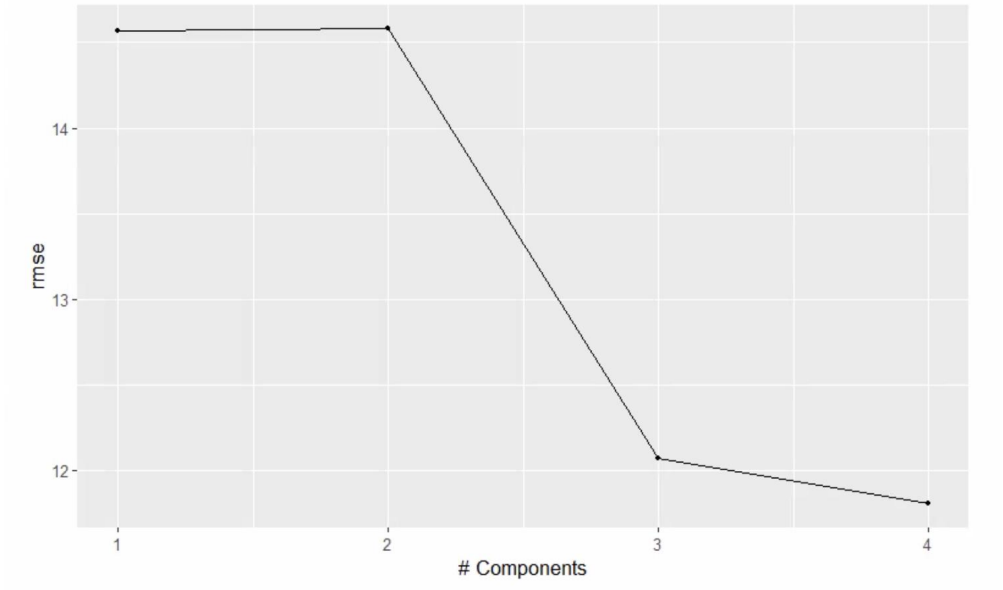
Summarizing the performance of each model:

Model Identifier	Metric Score	SE of Metric
linear_reg() step_pca(all_predictors(), num_comp = tune())	11.80477	0.2073285
linear_reg() step_YeoJohnson(all_predictors())	14.85359	3.026755
rand_forest()	8.490421	0.1308223
rand_forest()	8.499208	0.1322694
boost_tree()	7.897124	0.10262061
boost_tree()	7.902655	0.10297443
svm_rbf()	14.12467	0.1762308
nearest_neighbor()	9.783966	0.1608243

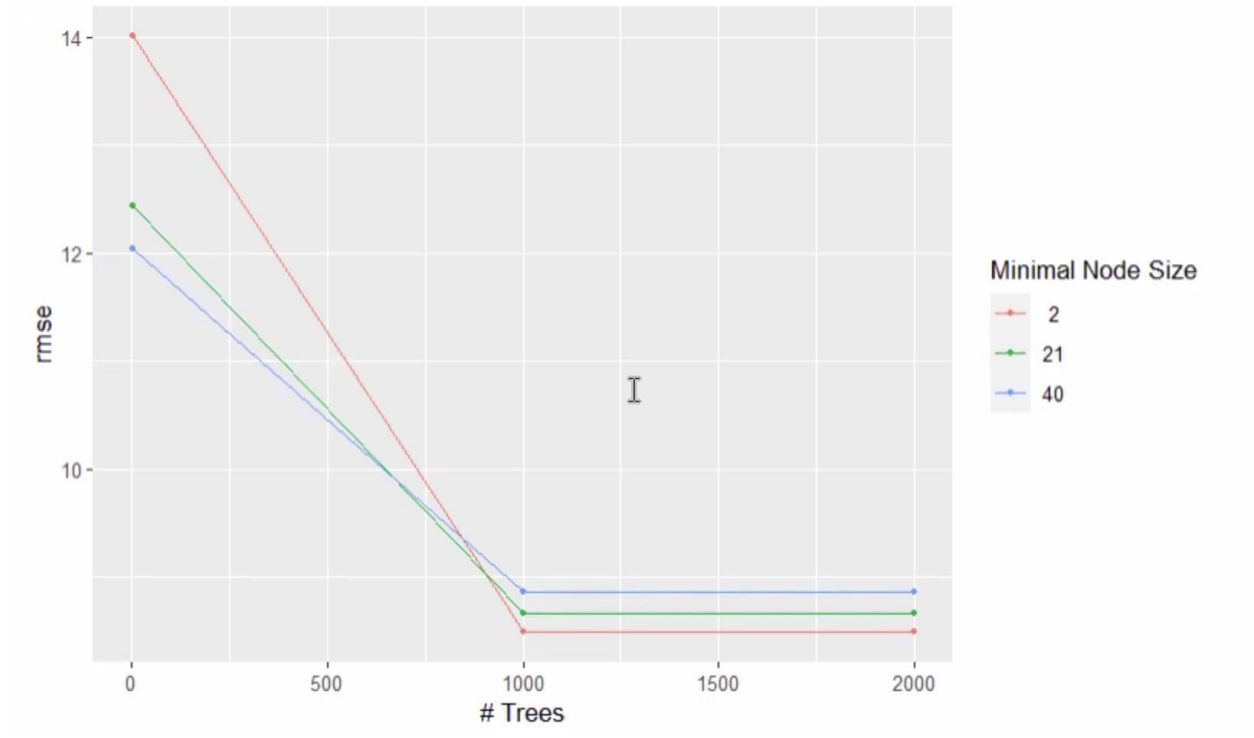
* Recipes are identical to the code in “Preprocessing / Recipes” Section

Autoplot for Comparing the performance of the different Models:

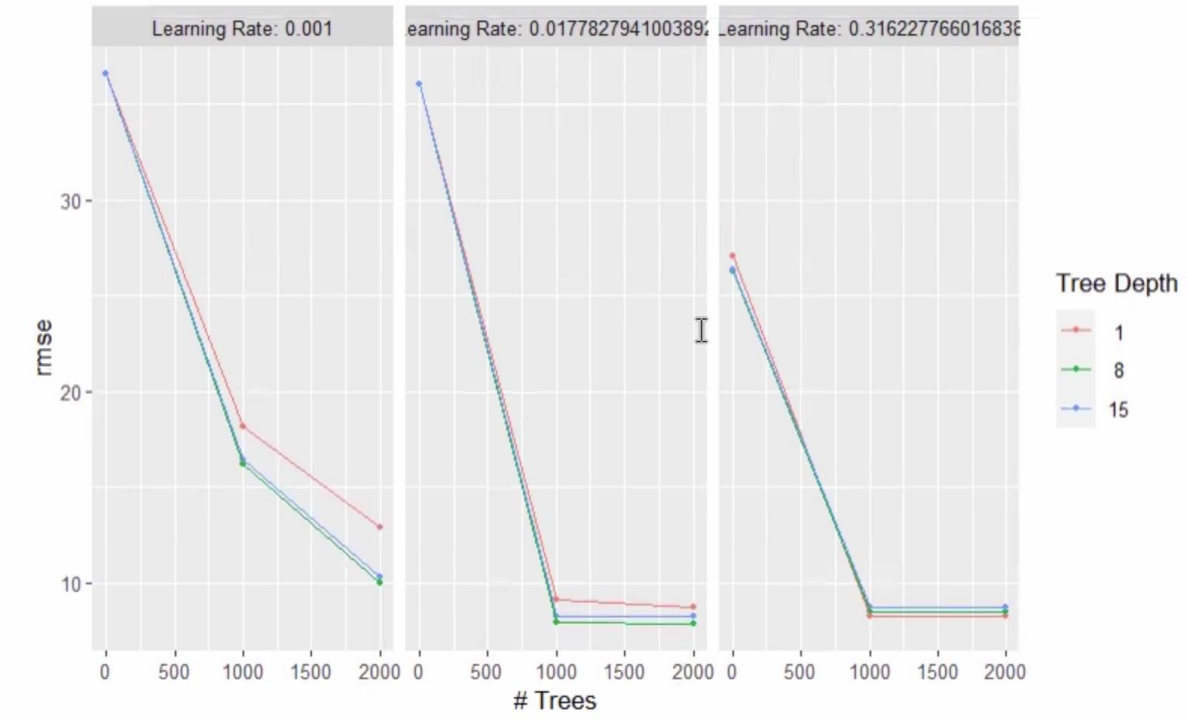
PCA:



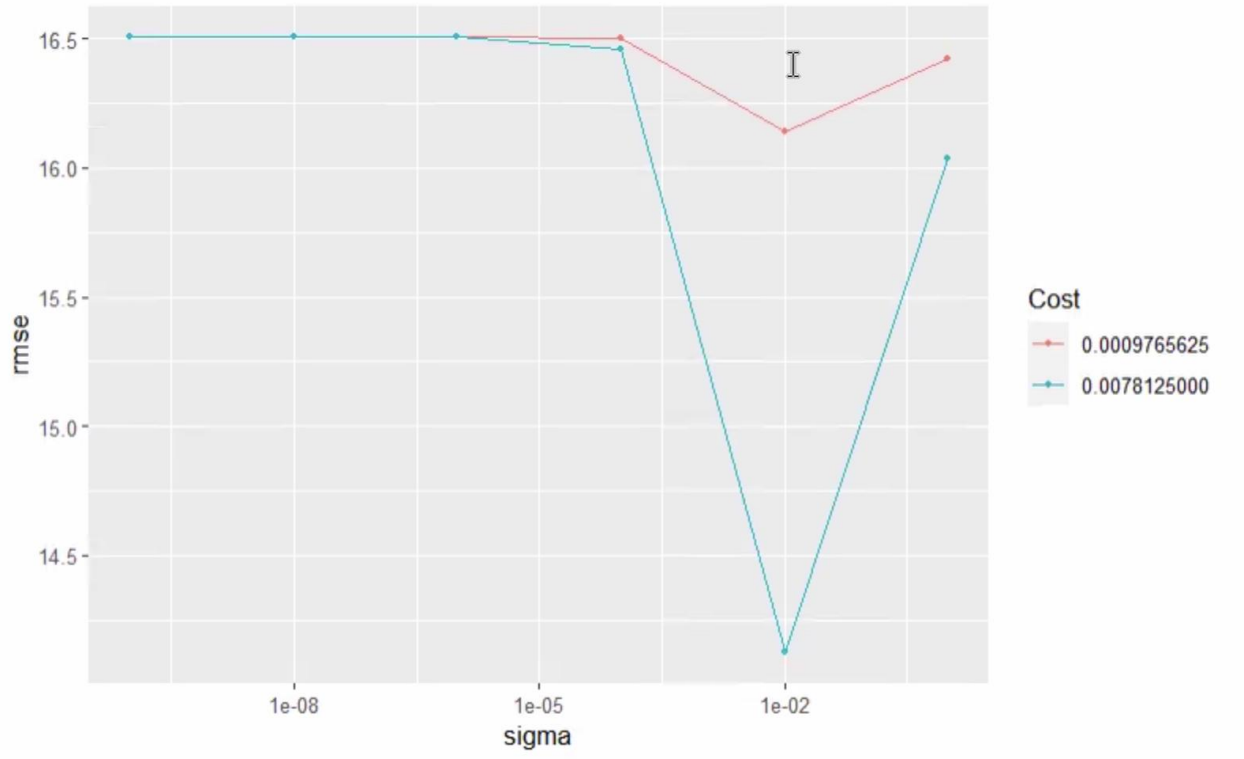
Random Forest:



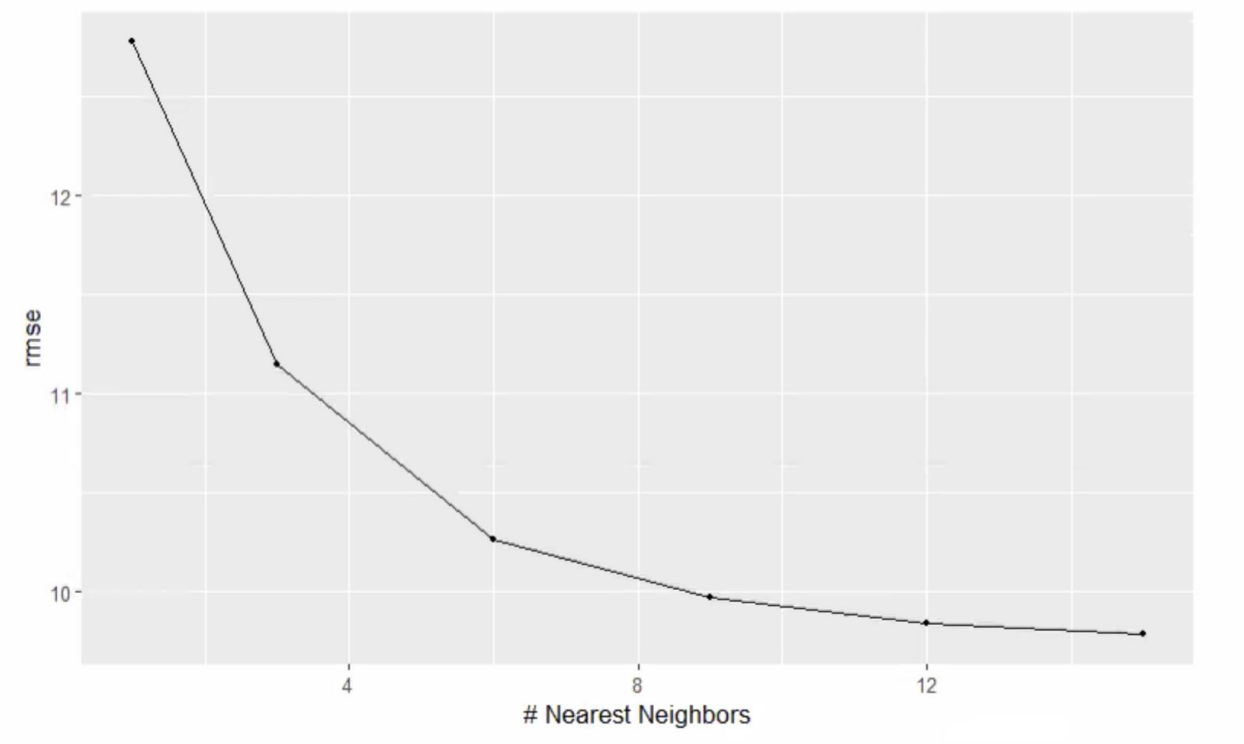
Boost Tree:



SVM:


16.5
16.0
15.5
15.0
14.5
rmse
1e-08
1e-05
1e-02
sigma
Cost
0.0009765625
0.0078125000

KNN:


12
11
10
rmse
4
8
12
Nearest Neighbors

Discussion of Final Model

For our final model, we have chosen to assemble a stack model comprising the following models: spline, boost tree, random forest, K-nearest neighbors (KNN), and support vector machine (SVM). We selected these models because they demonstrated lower RMSE values when they were evaluated individually. To further improve the performance of this stacked model, we decided to tune the hyperparameters of all models except for the spline model. From a pool of 55 potential candidates, the ensemble has chosen to include only 9 members. The nine highest weighted members are:

A tibble: 9 × 3

member <chr>	type <chr>	weight <dbl>
xgboost_res_1_09	boost_tree	0.363966730
xgboost_res_1_15	boost_tree	0.321683810
xgboost_res_1_17	boost_tree	0.132130948
knn_tune_1_3	nearest_neighbor	0.111911109
random_forest_res_1_2	rand_forest	0.057976312
xgboost_res_1_26	boost_tree	0.047350353
xgboost_res_1_27	boost_tree	0.008322109
xgboost_res_1_18	boost_tree	0.007198633
random_forest_res_1_4	rand_forest	0.000938686

9 rows

Given that our test dataset lacks a response variable, we executed the testing process by partitioning the training data into an 80% segment designated for training purposes and a 20% segment allocated for testing. We trained our stacked model using the 80% training data and subsequently evaluated its performance on the remaining 20% training data. This evaluation resulted in a RMSE of approximately 7.298129.

As a stack model, it boosts prediction quality by blending the strengths of many base models. Furthermore, this stacked model provides excellent flexibility due to the freedom in choosing which models to incorporate as base models. Additionally, we have the option to tune our models' hyperparameters and implement cross-validation techniques to prevent overfitting. However, this model has some drawbacks, primarily related to its computational demands. Specifically, it tends to require an extended runtime (Typically longer than one hour) due to the extensive tuning processes inherent in its constituent models. In addition, the increase in flexibility is associated with the decrease in interpretability.

There are several ways we can enhance our model's performance. Firstly, we can enhance our hyperparameter tuning process by setting a range of values and adding more levels, so we can have more values to maximize the performance of the values. Currently, for the random forest and boosted tree models, we only consider three levels for the number of trees (1, 1000, and 2000). Another way that could possibly improve our model's performance is to improve our recipe. We can add some interactions between the variables and explore a better way to determine which values are significant.

Appendix A: Final Annotated Script

Youtube Link for Script Verification Recording: https://youtu.be/cC1YFSSye4s

```
```{r}
# Load the library
library(tidymodels)
library(tidyverse)
library(gbm)
library(xgboost)
library(stacks)

# Import data
# Remove the predictor "name" since it is only for reference
train <- read_csv("train.csv") %>%
  select(-name)

set.seed(1)
# split the train data further into 80% train set and 20% test set (so we can test the stack model by computing the rmse)
train_split <- initial_split(train, prop = 0.8)
train1 <- training(train_split)
test1  <- testing(train_split)

# Import test data
test <- read_csv("test.csv")

# Fold the training data into a 10-fold cross-validation set. Stratify on `percent_dem`.
train_folds <- vfold_cv(train, v = 10, strata = percent_dem)
```
```

```
# The basic recipe of the models
basic_recipe <-recipe(percent_dem ~ id + x0001e + x0002e + x0003e +
x0022e + x0024e + x0034e + x0035e +  x0037e + x0038e + x0039e + x0044e
+ x0052e + x0058e + x0059e + x0060e + x0061e + x0062e + x0057e +
x0071e + x0086e + c01_006e + c01_007e + c01_008e + c01_009e + c01_010e
+ c01_011e + c01_012e + c01_013e + c01_014e + c01_015e +
income_per_cap_2016 + income_per_cap_2017 + income_per_cap_2018 +
income_per_cap_2019 + income_per_cap_2020 + gdp_2016 + gdp_2017 +
gdp_2018 + gdp_2019 + gdp_2020 + x2013_code, data = train) %>%
  # convert some columns as ratio and mutate them as new columns
  step_mutate(
    male_perc = x0002e / x0001e,
    female_perc = x0003e / x0001e,
    adult_pop = x0022e - x0024e,
    elder_pop = x0024e,
    white_race_perc = x0037e / x0001e,
    black_race_perc = x0038e / x0001e,
    american_race_perc = x0039e / x0001e,
    asian_race_perc = x0044e / x0001e,
    hawaiian_race_perc = x0052e / x0001e,
    other_race_perc = x0057e / x0001e,
    hisp_race_perc = x0071e / x0001e,
    mix_wb_perc = x0059e / x0001e,
    mix_wam_perc = x0060e / x0001e,
    mix_wai_perc = x0061e / x0001e,
    mix_bam_perc = x0062e / x0001e,
    high_school = c01_014e - c01_015e,
    bachelor_over = c01_015e,
    high_school_under = c01_006e - c01_014e) %>%
  # remove the original columns and the id column
  step_rm(
    x0001e, x0002e, x0003e, x0024e, x0022e, x0034e, x0035e, x0037e ,
x0038e , x0039e , x0044e , x0052e, x0057e, x0071e , x0059e , x0058e,
x0060e , x0061e , x0062e , c01_014e , c01_006e, c01_015e) %>%
  step_rm(id) %>%
  # remove rows with missing values in percent_dem
  # fill in missing values using bagged tree models for predictors
  step_naomit(percent_dem) %>%
  step_impute_bag(all_predictors()) %>%
  # remove predictors with low variance
  step_nzv(all_predictors()) %>%
  # create dummy variables for categorical variables
  step_dummy(all_nominal()) %>%
  # standardize numeric variables
```

```
  step_normalize(all_numeric_predictors())

# recipe for spline model
spline_rec <- basic_recipe %>%
  #removing high correlated variables
  step_corr(all_numeric(), -all_outcomes()) %>%
  # creating linear combination of numeric features
  step_lincomb(all_numeric(), -all_outcomes()) %>%
  # removing categorical variables
  step_rm(all_nominal()) %>%
  # box-cox transformation and Yeo Johnson transformation on
predictors
  step_bs(all_predictors()) %>%
  step_YeoJohnson(all_predictors())

# recipe for svm model
svm_rec <- basic_recipe %>%
  step_corr(all_predictors())

# create models
# total of 5 models: spline, random forest, boost tree, svm, knn
# for random forest, boost tree, svm, knn models,  we will tune their
hyperparameters
spline_model <- linear_reg() %>%
  set_mode("regression") %>%
  set_engine("lm")

random_forest_model <- rand_forest(min_n = tune(), trees = tune()) %>%
  set_mode("regression") %>%
  set_engine("ranger")

xgboost_model <- boost_tree(learn_rate = tune(), trees = tune(),
tree_depth = tune()) %>%
  set_mode("regression") %>%
  set_engine("xgboost")

svm_model <-
  svm_rbf(
    cost = tune("cost"),
    rbf_sigma = tune("sigma")
  ) %>%
  set_engine("kernlab") %>%
  set_mode("regression")

knn_model <-
```

```
  nearest_neighbor(
    mode = "regression",
    neighbors = tune("k")
  ) %>%
  set_engine("kknn")


# Create workflows for all the models
spline_wf <- workflow() %>%
  add_model(spline_model) %>%
  add_recipe(spline_rec)

random_forest_wf <- workflow() %>%
  add_model(random_forest_model) %>%
  add_recipe(basic_recipe)

xgboost_wf <- workflow() %>%
  add_model(xgboost_model) %>%
  add_recipe(basic_recipe)

svm_wflow <-
  workflow() %>%
  add_model(svm_model) %>%
  add_recipe(svm_rec)

knn_wflow <-
  workflow() %>%
  add_model(knn_model) %>%
  add_recipe(basic_recipe)


# set up parameter grid for those four models
# random_forest and boost_tree: for each hyperparameters, explore
three different values
ranforest_grid <- parameters(random_forest_model) %>%
  grid_regular(levels = 3)

xgboost_grid <- parameters(xgboost_model) %>%
  grid_regular(levels = 3)

# svm and knn: for each hyperparameters, explore six different values
svm_grid <- grid_regular(parameters(svm_model), levels = 6, filter =
c(cost <= 0.05))

knn_grid <- grid_regular(parameters(knn_model), levels = 6)
```

```r
# fit
# save predictions and workflow objects during the tuning process
ctrl_grid <-  control_grid(save_pred = TRUE, save_workflow = TRUE)
ctrl_res <- control_grid(save_pred = TRUE, save_workflow = TRUE)
metric <- metric_set(rmse) # for evaluating the performance

# Tuning process
spline_res <- spline_wf %>%
  fit_resamples(
    resamples = train_folds,
    metrics = metric,
    control = ctrl_res
  )

random_forest_res <- random_forest_wf %>%
  tune_grid(
    resamples = train_folds,
    grid = ranforest_grid,
    metrics = metric,
    control = ctrl_grid
  )

xgboost_res <- tune_grid(
  xgboost_wf,
  resamples = train_folds,
  grid = xgboost_grid,
  metrics = metric,
  control = ctrl_grid
)

svm_tune <- tune_grid(
  object = svm_wflow,
  resamples = train_folds,
  grid = svm_grid,
  control = ctrl_grid,
  metrics = metric
)

knn_tune <- tune_grid(
  object = knn_wflow,
  resamples = train_folds,
  grid = knn_grid,
  control = ctrl_grid,
```

```
  metrics = metric
)


# stacks multiple predictive models (the five we have)
ames_stack <- stacks() %>%
  add_candidates(spline_res) %>%
  add_candidates(random_forest_res) %>%
  add_candidates(xgboost_res) %>%
  add_candidates(svm_tune) %>%
  add_candidates(knn_tune)

# combining predictions from models in stacks
# then fit the blended predictions to create an ensemble model
ames_stack <- ames_stack %>%
  blend_predictions() %>%
  fit_members()

#  make prediction on the test data
#  bind the predicted values to the id column in the test data
prediction <- ames_stack %>%
   predict(new_data = test) %>%
   bind_cols(id = test$id)

# exporting prediction as csv file for Kaggle submission
prediction %>%
   rename(percent_dem = .pred) %>%
   write_csv("predictions_last2.csv")

```

Appendix B: Team Member Contributions

Zhuowei Deng:

- Tuning the hyperparameters of all the models
- Creating and testing of candidate models
- Creating and testing the stack models
- Creating the recipes
- Report Writing: Appendix A, Discussion of Final Model, Candidate Models/ Model Evaluation/ Tuning

Liaohan Wang

- Report Writing : Exploratory Data Analysis visualizations and interpretations
- Report Writing : Introduction
- Testing models

Yifan Jiang

- Testing and fixing of recipes/models
- Report Writing: Exploratory Data Analysis
- Report Writing: Visualizations

Tou-Chia Chang

- Creating and testing of candidate models
- Creating and testing the stack models
- Creating the recipes
- Report Writing: Preprocessing and Recipes