2020 Presidential Election

Regression Project Report

Summer 2022

Introduction to Statistical Models and Data Mining

Prof. Miles Chen

July 29, 2022

Team: YYDS

Jingyi Lang

Zhitong Zhou

Olivia Wang

Yanjun Jiang

Introduction

This report presents a process of creating a high performing supervised learning regression model that uses a given training data set and a test data set on RStudio, which contains 214 predictors showing the predicted demographic and education information for voters in each county. The response variable is the percentage of voters in a county that voted for Biden in the 2020 US Presidential Election. We aim to fit a regression model that produced the lowest rmse for the test data set (Root Mean Square Error).

Based on the data description[1] and an analysis report for Biden's 2020 Victory by Pew Research Center[2]. Our team supposes that the variables related to age between 18 and 49 may be associated with the response variable because the supporting percentage points favoring Joe Biden were significantly higher than the percentage points of Trump among the voters of these age groups. Moreover, according to the “wide educational differences among 2020 voters” graph of the analysis report, Biden also got higher percentage points among the population that receives a higher level of education (College Graduate and Postgraduate), which made us think that the variables with a higher level of education may be associated with the response variable.

References

[1] UCLA Stats 101C 22Summer, Course competition. *Kaggle.* https://www.kaggle.com/competitions/ucla-stat-101c-22summer/data.

[2] Igielnik, R., Keeter, S., & Hartig, H. (2021, June 30). Behind Biden’s 2020 Victory. *Pew Research Center*. https://www.pewresearch.org/politics/2021/06/30/behind-bidens-2020-victory/.

[3] Robertson, T. (2020, December 16). What Is Data Normalization? Why Is it Necessary?. *DatascienceAcademy.io*. https://www.datascienceacademy.io/blog/what-is-data-normalization-why-it-is-so-necessary/.

[1] Reference [1]
[2] Reference [2]

[4] XGBoost R Tutorial. https://xgboost.readthedocs.io/en/stable/R-package/xgboostPresentation.html.

[5] Kumar, Dinesh. (2021, December 26). A Complete understanding of LASSO Regression. *Great Learning*. https://www.mygreatlearning.com/blog/understanding-of-lasso-regression/#:~:text=What%20is%20Lasso%20Regression%3F%20Lasso%20regression%20is%20a,over%20regression%20methods%20for%20a%20more%20accurate%20prediction.

[6] Regression Example with XGBoost in R (2019). *DataTechNotes*. https://www.datatechnotes.com/2020/08/regression-example-with-xgboost-in-r.html.

[7] Enders, B. F. Collinearity. *Britannica*. https://www.britannica.com/topic/collinearity-statistics

[8] Log Transformation. *Quantargo*. https://www.quantargo.com/courses/course-r-machine-learning-tidymodels/02-preprocessing/03-data-transformation/section-log-transformation

[9] Interpretation of the Principal Components. The Pennsylvania State University. https://online.stat.psu.edu/stat505/lesson/11/11.4

Exploratory Data Analysis

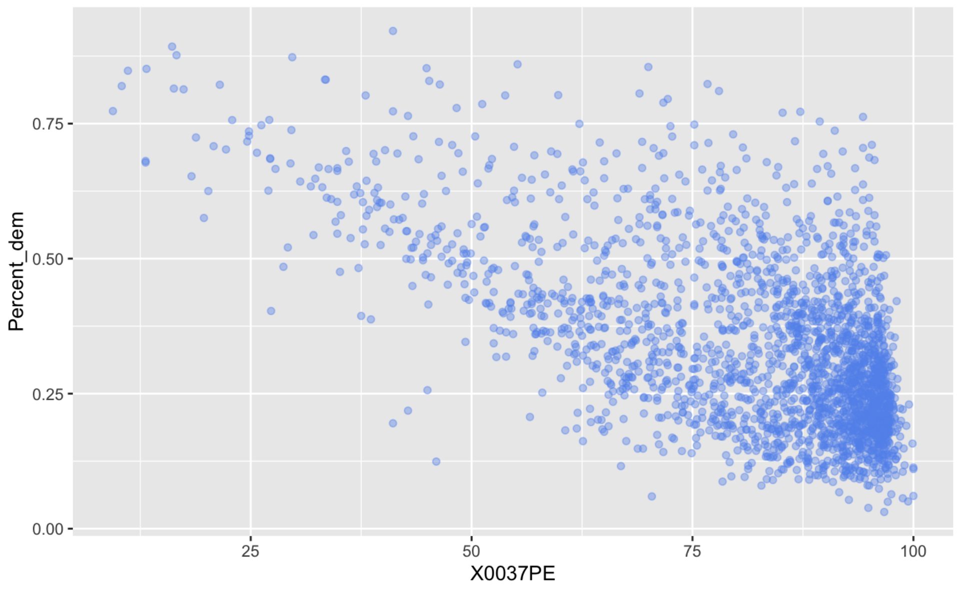
1. Drop duplicate columns and missing observations

When we first got the data set, we scanned through the columns and noticed some duplicated columns in the data. Hence, the first thing we did was remove the duplicated columns. Next, we also noticed some missing observations in the dataset, which will affect our further operations on the data, so we removed the observations with NAs. The new training data after basic data cleaning contain 2330 observations and 214 variables.

2. Scatterplot

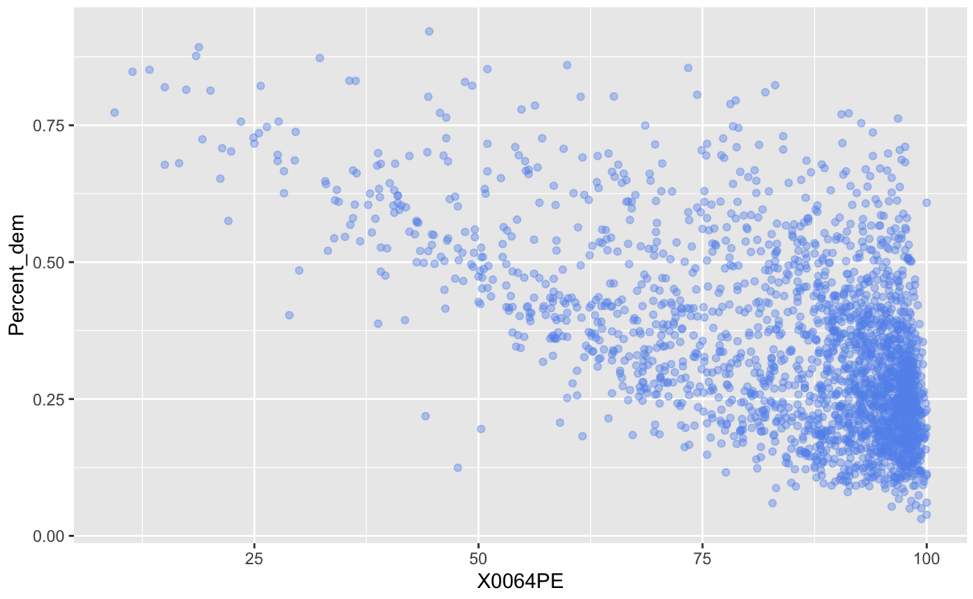
We use scatter plots to explore the potential relationships between the predictors against the response variable. Out of the 214 plots, there are some plots showing a clear pattern, which are:

- *X0037PE (Percent_Total population_One race_White) vs. Percent_dem*



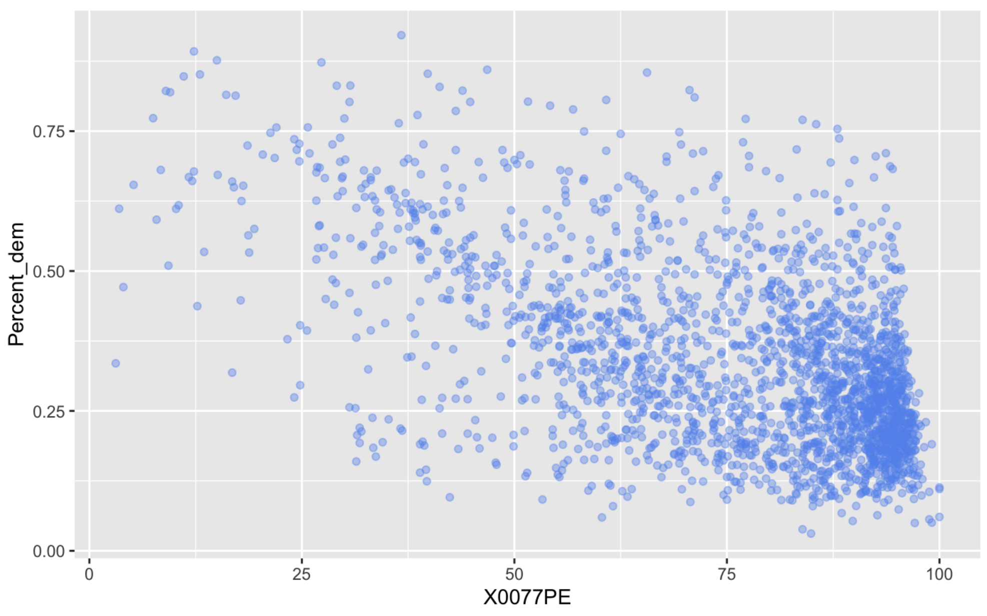
Above is a scatter plot of the predictor *X0037PE* (on the x-axis) and the response *Percent_dem* (on the y-axis). While *X0037E* increases, *Percent_dem* decreases, so it shows a negative relationship between the percent of population with only white races in each county and the percent of voters who voted for Biden in each county.

- *X0064PE (Percent_Race alone or in combination with one or more other races_Total population_White) vs. Percent_dem*



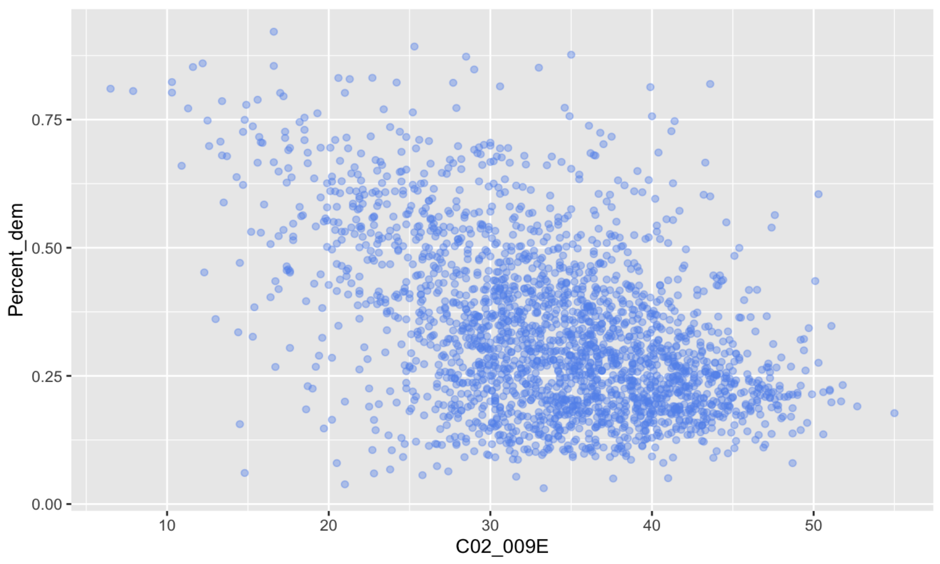
Above is the scatter plot showing the relationship between the variable *X0064PE* and the response variable *Percent_dem*, which indicates an obvious negative pattern that as the variable, *Percent_Race alone or in combination with one or more other races_Total population_White*, increases, the *Percentage of voters in a county that voted for Biden* decreases.

- *X0077PE (Percent_Total population_Not Hispanic or Latino_White alone) vs. Percent_dem*



This scatter plot also shows a clear relationship between the variable *X0077PE* and the response variable *Percent_dem*. As shown above, while *Percent_Total population_Not Hispanic or Latino_White alone* grows up, the *Percentage of voters who voted for Biden* decreases, which indicates a decreasing pattern, same as a negative relationship, between the predictor and the response.

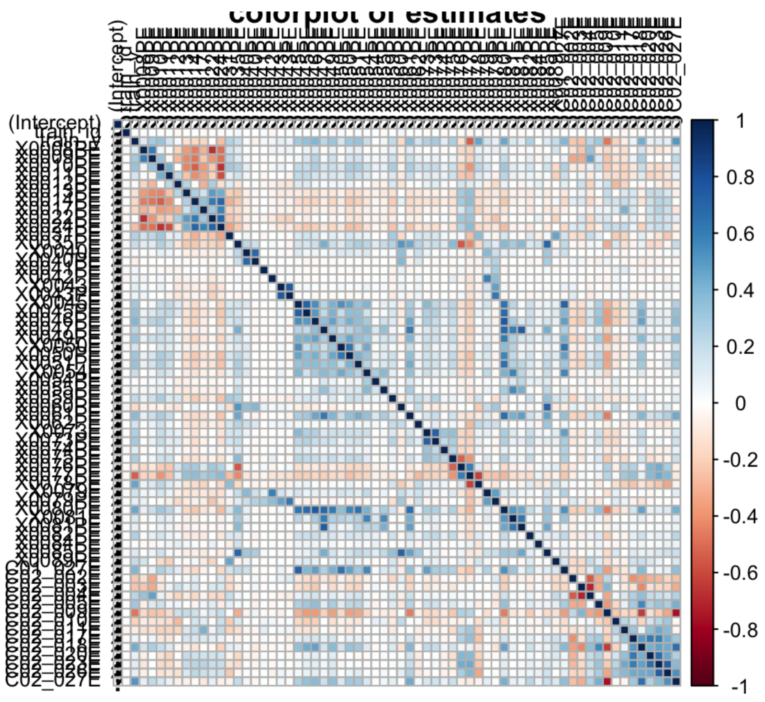
- *C02_009E (Estimate_Percent_Population 25 years and over_High school graduate (includes equivalency)) vs. Percent_dem*



The plot of *Percent_dem* vs *C02_009E* shows the relationship between the response *Percent voters who voted Biden* and the variable *Estimate_Percent_Population 25 years and over_High school graduate (includes equivalency)*. The plot shows a decreasing pattern which indicates there are potential relationships between these responses and predictors. As the value of Estimate_Percent_Population 25 years and over_High school graduates increases, the response Percent voters who voted Biden decreases.

3. Check correlation between predictors

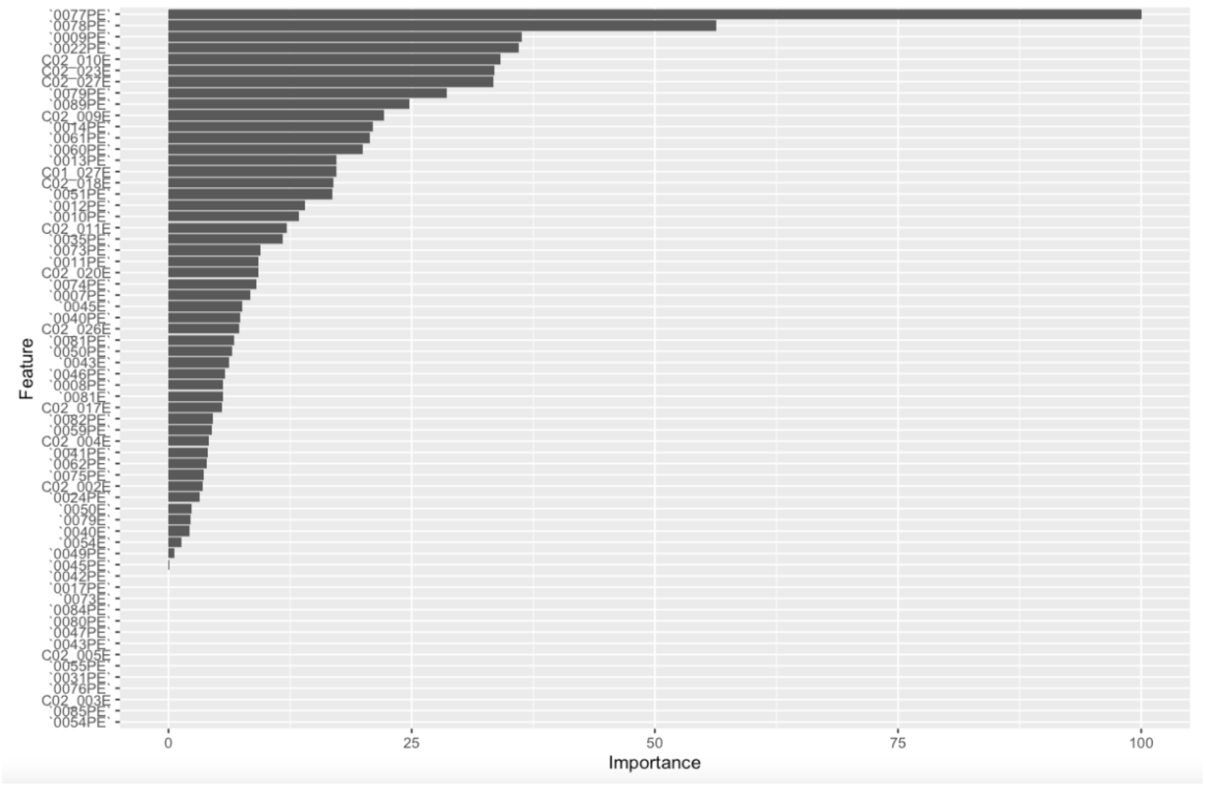
We then decide to check for collinearity between predictor variables because when predictor variables in the same regression model are correlated, they cannot independently predict the value of the dependent variable. In other words, they explain some of the same variances in the dependent variable, reducing their statistical significance. After running the cor() function on our predictors, we decide to remove variables with a correlation absolute value bigger than 0.8 since we have a relatively big sample size.



The above correlation plot shows the correlation between variables after removing collinearity above 0.8; From the plot, we can see most of the severe collinearity has been removed from our data. The predictors left have relatively mild collinearity problems.

4. LASSO Selection and log transformation

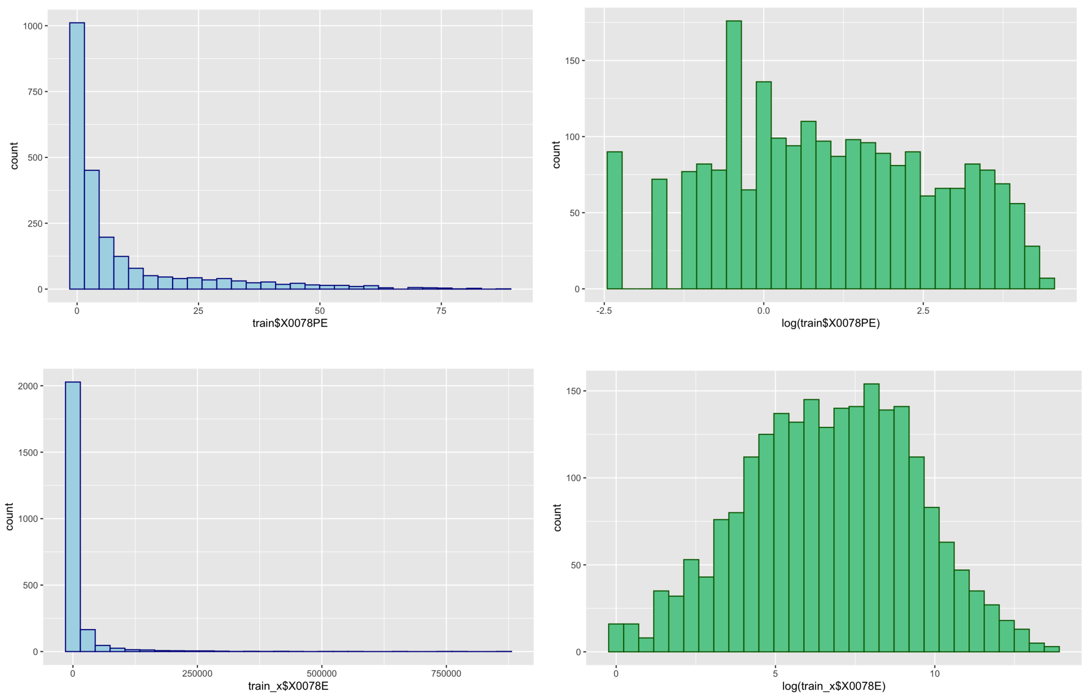
After removing correlations, we tried a few variable selection methods. The first one is LASSO selection. Lasso selection can identify the predictors that are not strongly associated with the response variable. Then, Lasso forces the coefficients of the variables towards zero. Using this method we plot an important descending plot of the predictors, then we remove 14 predictors that are the least associated with the predictors. According to the plot and the coefficients, *0077PE, 0078PE, 0009PE, 0022PE, C02_010E* are the top 5 significant variables. We visualize these variables and find out that the histogram and notice the distributions of these predictors are highly skewed. For these cases, data transformation steps provide a way for creating more favorable distributions while keeping the information content of the variable intact. After applying the log transformation[3], the variables have a much nicer bell-shaped distribution.



The above graph shows the descending ranking of variables' importance after the LASSO method applied. The last 14 variables are the least important ones and we assume they will not make significant contributions to our model so we can take them out from the potential predictors. And the variables on the top of the graph are the most important ones

[3] Reference [8]

so we decide to perform log transformation on it and see if the transformation can improve model accuracy.



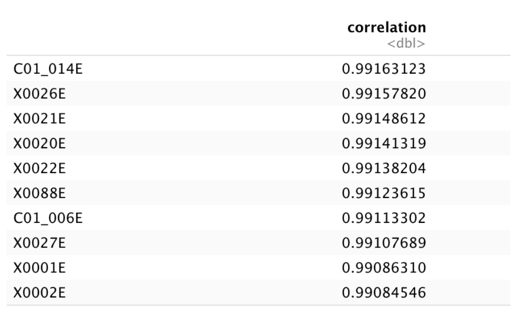
The above graphs are examples of the frequency distribution of the variables before and after log transformation. The plot in blue is the original distribution of the variable, and the green graph is the distribution after log transformation. The plot shows that the variables have a more excellent bell-shaped distribution.

5. PCA method variable selection using correlation[4]

The other selection method we tried is the Principle Component Analysis; the goal of it is to summarize the correlations among our predictors with a smaller set of linear combinations. We first compute the correlations between our response variable and each principal component.

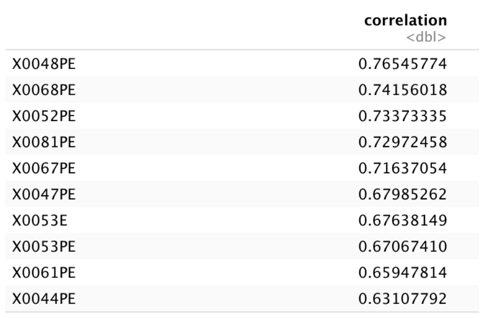
[4] Reference [9]

Dim 1



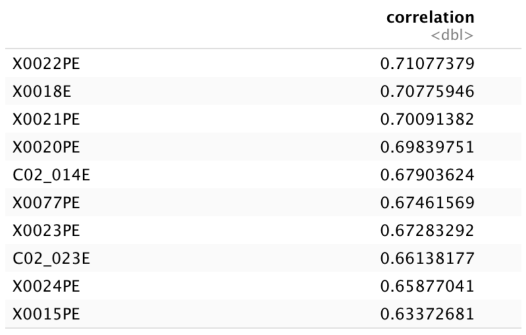
	correlation <dbl>
C01_014E	0.99163123
X0026E	0.99157820
X0021E	0.99148612
X0020E	0.99141319
X0022E	0.99138204
X0088E	0.99123615
C01_006E	0.99113302
X0027E	0.99107689
X0001E	0.99086310
X0002E	0.99084546

Dim 2



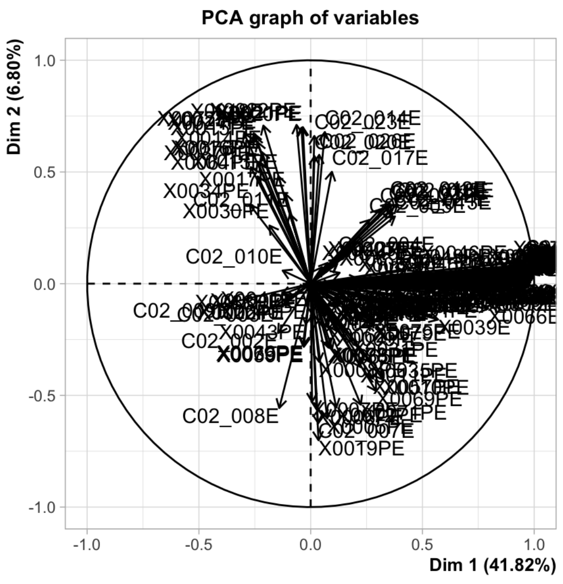
	correlation <dbl>
X0048PE	0.76545774
X0068PE	0.74156018
X0052PE	0.73373335
X0081PE	0.72972458
X0067PE	0.71637054
X0047PE	0.67985262
X0053E	0.67638149
X0053PE	0.67067410
X0061PE	0.65947814
X0044PE	0.63107792

Dim 3



	correlation <dbl>
X0022PE	0.71077379
X0018E	0.70775946
X0021PE	0.70091382
X0020PE	0.69839751
C02_014E	0.67903624
X0077PE	0.67461569
X0023PE	0.67283292
C02_023E	0.66138177
X0024PE	0.65877041
X0015PE	0.63372681

The principal components (Dim_1) are strongly correlated with many of the original variables. The top 5 are: *C01_014E, X0026E, X0021E, X0020E, X0022*E. The first principal component will increase with there is an increase of these variables . This suggests that these five criteria vary together. If one increases, then the remaining ones tend to increase as well.



The above graph shows which variables correlate most strongly with each principal component. Since the determination of what level of correlation is considered important is a very subjective decision, we tried many combinations of dimension correlation numbers. We chose 0.2 for the first dimension and 0.6 for the second dimension to keep roughly 150 predators for fitting models.

6. Check Interactions among variables

ANOVA Table for Interaction terms

```
X0011PE:X0016PE      1  0.202  0.2021  11.919 0.0005656 ***
X0022PE:X0057PE      1  0.210  0.2103  12.402 0.0004371 ***
X0051PE:X0078E       1  0.670  0.6702  39.528 3.853e-10 ***
X0051PE:C02_007E     1  0.671  0.6706  39.551 3.807e-10 ***
X0022PE:C02_007E     1  0.516  0.5155  30.406 3.894e-08 ***
```

In order to fit a better model, our team considered finding the interactions among variables, After using linear regression model function *lm()* to fit the interaction terms, the ANOVA table shows that the interactions *X0011PE:X0016PE, X0022PE:X0057PE, X0051PE:X0078E, X0051PE:C02_007E*, and *X0022PE:C02_007E* are significantly affect the response. Therefore, we considered using these interactions in our model in our recipe.

Recipes

For the recipes we used for our models, since our candidate models are classification models which require recipes, we created three recipes to check if there would be significant differences or improvements in our predicted test results.

1. **Basic recipe** (code *reci*), which used *recipe()* function to create a basic recipe for the models we used. Since in our training data, there is a column named *train_id*, declaring the id numbers of the training data, it is not related to the response variable. So we add a step *update_role(train_id, new_role = "ID")* to tell the recipe the role of this variable is just an id column.

2. **Logarithmic transformation recipe** (code *log_rec*) , which adds *step_log(all_numeric_predictors(), base = 10, signed = TRUE)* function to log transform data in the basic recipe. Because based on the exploratory process, we found that some numeric predictors are highly skewed, so we decided to add log transformation in the recipe to check if there would be any improvements for the test results.

3. **Normalized recipe** (code *norm_reci*), which adds *step_normalize(all_numeric_predictors()*) to normalize all numeric predictors because we expect to increase coherence of our data and gets rid of a variety of irregularities that can make it more difficult to interpret the data[5].

4. **Interaction recipe** (code new_reci), which adds *step_interact(terms = ~ X0051PE:X0077PE + X0022PE:C02_007E + X0011PE:C02_023E)* to add some two-variables interaction terms in the basic recipe. Since when we created a simple linear regression model for the remaining variables after checking the importance level of the original variables, we selected 11 variables with a significance level of three '***' in the summary table and created interaction terms in a new linear regression model for these variables, and we found that some interactions are significant in the anova table. Thus, we selected three interaction terms with the most significance level to check if there would be any improvements for our model.

[5] Reference [3]

Candidate Models

In order to avoid the overfitting issue, our team decides to test the performance of each model by using cross-validation on the training data set. The cross-validation is to partition the data into v equally sized blocks, then set one block as the validation set, while the remaining blocks combine to be the training data. We divide the data into 8 cross-validation folds for evaluating our models' performance. The table below shows the 6 candidate models we decided to test.

Model Identifier	**Type of Model**	**Engine**	**Recipe Used**	**Hyperparameters**
lasso_model_tuning	LASSO	glmnet	log_rec	● penalty = tune() ● mixture = 1
ridge_model_tuning	Ridge Regression	glmnet	log_rec	● penalty = tune() ● mixture = 0
knn_model_tuning	KNN	kknn	reci	● neighbors(range(1, 100)) ● levels = 10
elastic_model_tuning	Elastic Net	glmnet	log_rec	● penalty = tune() ● mixture = tune()
mod_rf	Random Forest	ranger	reci	● mtry = tune() ● min_n = tune()
xgb_predict_var149	XGBoost[6]	xgboost	reci	● trees = 1000 ● tree_depth = tune() ● min_n = tune() ● loss_reduction = tune() ● sample_size = tune() ● mtry = tune() ● learn_rate = tune()
xgb_predict_var80	XGBoost	xgboost	reci	● trees = 1000 ● tree_depth = tune() ● min_n = tune() ● loss_reduction = tune() ● sample_size = tune() ● mtry = tune() ● learn_rate = tune()

[6] Tutorial Reference [4]

- **Model: *lasso_model_tuning***

  As a regularization technique, our team firstly employed LASSO regression to create a model[7]. We firstly create a lasso model with *linear_reg(penalty = tune(), mixture = 1)* and *set_engine("glmnet")* functions and add the model and the recipes to the workflow, then we set up the grid with *grid_regular(penalty(range(-2, 2)), levels = 100)*. Finally we piped the workflow into the *tune_grid()* function with the resamples equals to the 8 v-folds and the grid. We performed this lasso model with basic recipe, logarithmic transformation recipe, and normalized recipe respectively to evaluate which recipes would help the model perform better.

- **Model: *ridge_model_tuning***

  We performed a ridge regression model as a shrinkage penalty method, similar to the LASSO regression, with *linear_reg(penalty = tune(), mixture = 0)* and *set_engine("glmnet")* functions and add the ridge regression model and the recipes to the workflow. Then we set up the grid with the same grid as the LASSO model and fitted the model with the workflow and *tune_grid ()* function with the grid and the v-folds as inputs. Finally we attempted the different recipes respectively to check which recipe improved the model.

- **Model: *knn_model_tuning***

  We also attempted a knn model to see the performance of the model on this training data set. We set up the knn model with *nearest_neighbor()* function by tuning the parameter "neighbors". And we chose the grid of knn with *grid_regular(neighbors(range(1, 100)), levels = 10)*. Similar to the process of fitting the linear regression models, we fitted the knn model with workflow and the tuning grid. After fitting the knn model, we attempted the different recipes to measure their performances.

- **Model: *elastic_model_tuning***

[7] Reference [5]

After we fitted the LASSO and Ridge regression models, we also expected to see the performance of the elastic-net model, which is a hybrid of LASSO and Ridge regression. Different from setting the "mixture" argument inside the *linear_reg()* function like LASSO or Ridge model, we used the *tune()* function for the "penalty" and "mixture" arguments and set the engine as "glmnet." Then we used the similar process as the candidate models above to fit the elastic-net model and tried different recipes to measure their performances.

- **Model:** ***mod_rf***

  Random forest regression model is another popular regression which can deal with the problem if the data that are overly correlated. We created a random forest model with the *rand_forest()* function with *mtry = tune(), min_n = tune(), and trees = tune()* arguments. After adding the model and the recipes to create the workflow, we randomly selected ranges for the "mtry," "trees," and "min_n" parameters in the *grid_regular()* function and resampled the v-folds in the *tune_grid()* to fit the model. Then we found that the best model was in the parameters of *mtry = 40, trees = 120, and min_n = 5*. So we reset the grid and with each parameter in the range including the best hyperparameters and used *select_best()* and *finalize_workflow()* commands to find the best set of hyperparameters to finalize the workflow. Finally, we used the *last_fit()* command to fit a best random forest model and attempted the candidate recipes to check the improvement of the model performance.

- **Model:** ***xgb_predict_var149***

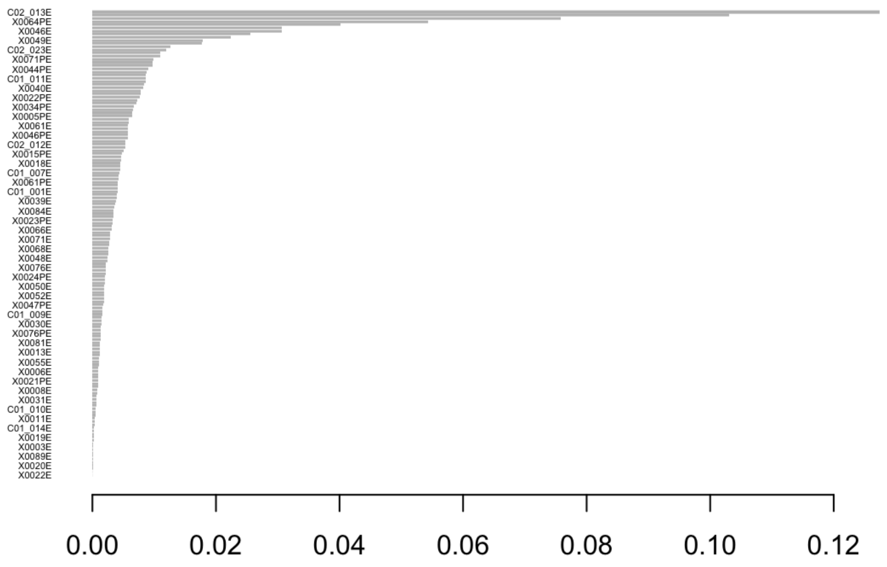
  Compared to random forest, XGBoost[8] is an implementation of gradient boosting trees algorithm with high predictive accuracy. After we performed the PCA method on the training data set, we create a XGBoost model with the remaining data of 149 variables, we tuned every parameter in the model except for setting *trees = 1000* with the *tune()* function when creating the model because if we used the *tune()* function in the *trees* parameter, the code would caused an error. Then we created a workflow for the model and used the *tune_grid()* function with setting "*grid =*" a set of empty parameters in the *grid_latin_hypercube()* function to get the XGBoost results and prepare for the next step of checking the different recipes.

[8] Reference [6]

- **Model:** ***xgb_predict_var80***

  After we created a XGBoost model with the 149 variables and the attempts of using the different recipes, in order to find a way to improve the model performance, we used the *data.matrix()* function on the data with 149 variables and *xgb.importance()* function on the last XGBoost model we fitted to evaluate the importance level of the 149 variables. Then we selected the most important 80 variables based on the importance level and created a new XGBoost model and attempted the candidate recipes to measure the model's improvement.

Model Evaluation and Tuning



For a better and clearer comparison among the different candidate models, we used the same **8 v-folds** generated by the cross-validation to measure the performance of the candidate models.

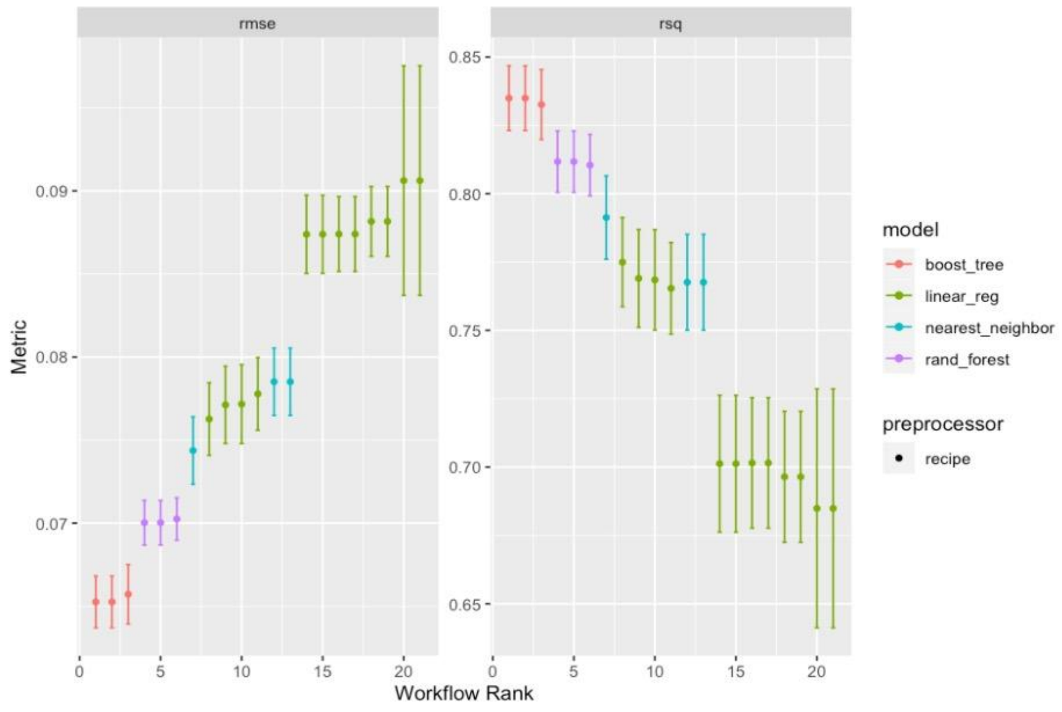
After turning the candidate models' hyperparameters respectively, we picked the parameter combination which generated the lowest rmse for each model. A lower rmse indicates that the predictions generated by the model are close to the actual value, which means the model's accuracy is high. Below are the performance of all regression models after turning and picking the best combination of hyperparameters.

- **Summary Table of the Performance of Each Model**

Model Identifier	**Metric Score (rmse)**	**Hyperparameters**	**SE of Metric**
lasso_model_tuning	0.08551552	● penalty = 0.01 ● mean = 0.08551552 ● n = 8	● std_err = 0.0009174638
ridge_model_tuning	0.0749889	● penalty = 0.01 ● mean = 0.0749889 ● n = 8	● std_err = 0.0007263957
knn_model_tuning	0.08234178	● neighbors = 12 ● mean = 0.08234178	● std_err = 0.001739935
elastic_model_tuning	0.07519589	● penalty = 0.01 ● mixture = 0 ● mean = 0.07519589 ● n = 8	● std_err = 0.0007920767
mod_rf	0.07308657	● n = 1 ● mean = 0.07308657	● std_err = 0.0009577267
xgb_predict_var149	0.06697	● trees = 1000, ● tree_depth = 8 ● min_n = 5 ● loss_reduction = 5.638207e-10 ● sample_size = 0.2351550, ● mtry = 131 ● learn_rate =	● std_err = 0.001289143

		0.004848720	
xgb_predict_var80	0.06518	● trees = 1000, ● tree_depth = 6, ● min_n = 11, ● loss_reduction = 2.166431e-10, ● sample_size = 0.3461942, ● mtry = 27, ● learn_rate = 0.021839515	● std_err = 0.001486692

- **Autoplot of All Models**



According to the Autoplot of all model performance, linear_reg shows the performance of the linear/lasso/ridge/elastic-net regression model (mean rmse around 0.09). The nearest-neighbor represents the performance of KNN regression model (mean rmse around 0.076). The rand_forest represents the performance of the random forest regression model (mean rmse around 0.07). The boost_tree represents the performance of the XG boost regression model (mean rmse around 0.065). Since our target is to find the model which generates the lowest rmse, we decided to use **XGBoost regression model** as our final model.

Final Model: XGBoost Regression Model with 80 Predictors

- **Final Model**

  xgb_predict_model <- boost_tree(trees = 1000, tree_depth = 8, min_n = 5, loss_reduction = 5.638207e-10, sample_size = 0.2351550, mtry = 131,learn_rate = 0.004848720) %>% set_engine("xgboost") %>% set_mode("regression")

The final model we used is XGBoost (Extreme Gradient Boosting) regression, it is constructed from decision tree models. The XGBoost regression is an efficient implementation of the gradient boosted trees algorithm. It attempts to predict a target variable accurately by combining the estimates of a set of simpler and weaker models. The XG Booster regression model we used has 7 hyperparameters: *trees, tree_depth, min_n, loss_reduction, sample_size, mtry,* and *learn_rate*.

- **Strength**
  1. The precision of the model outperformed all other candidate models.
  2. XGBoost regression is a popular supervised machine learning method with functions like parallelization, distributed computing, and performance.
  3. The XGBoost regression model has more hyper-parameters that can be tuned, which means that the model can be more accurate.

- **Weakness**
  1. Compared to the linear regression and KNN regression model, the XGBoost regression model needs more time to run.
  2. Have weaker performance on sparse and unstructured data.
  3. Harder to interpret compared to other models.

- **Possible improvements**

1. **Try a different set of variables.** Due to the time limit, we do not have enough time to run enough combinations of variables. There should be a combination of variables that can be used to generate a better model.
2. We can also **try more different recipes** for the model to improve the model.
3. **Adding more interactions.** Because of the large number of variables, we did not use all possible interaction among the variables. By adding more interactions, the performance of the model might be improved.
4. For the additional data, our team considers that **the Parties that the voters support**, such as Republican Party, Democratic Party, other parties, and Independents, **the religious groups that the voters belong to**, and **the marital status of the voters** might be useful.

Appendix: Final Annotated Script

~~~
### Loading the train and test data set
```{r}
train <- read.csv("train.csv")
test <- read.csv("test.csv")
```

### Removing duplicated columns
```{r}
index <- (1:215)[duplicated(as.list(train))]
dup_cols <- train[,index]
train <- train[!duplicated(as.list(train))]
```

### Drop observations with NAs
```{r}
library(tidyr)
train <- train %>% drop_na()

train_y <- train$percent_dem
train_id <- train$id
```

### Remove the response variable
```{r}
train_x <- train[ , -2]
```

### PCA method for variable selection
```{r}
library(FactoMineR)
~~~

```
pca <- PCA(train[, -c(1:3)], scale.unit = TRUE, ncp = 10, graph =
FALSE) #using the PCA function from the FactoMineR library to
perform PCA method on the training data
dimdesc(pca) -> vals # storing the correlation



## Check and remove correlations for each dimensions
df1 <- vals[["Dim.1"]][["quanti"]] %>% data.frame() %>%
filter(abs(correlation) >= .2)
df1$Variable = rownames(df1)

df2 <- vals[["Dim.2"]][["quanti"]] %>% data.frame() %>%
filter(abs(correlation) >= .6)
df2$Variable = rownames(df2)

df3 <- vals[["Dim.3"]][["quanti"]] %>% data.frame() %>%
filter(abs(correlation) >= .9)
df3$Variable = rownames(df3)

dat1 <- rbind(df1, df2, df3)

colmn <- which(names(train) %in% dat1[, 3])
PCA_new <- train[,colmn] # storing the new selected predictors
into a new data frame
PCA_new <- cbind(train_y,PCA_new)
```

### Creating folds and basic recipe

```{r}
# load needed libraries
library(xgboost)
library(rsample)
library(tidymodels)
```

~~~
set.seed(10) # set seed for reproducing results

folds <- vfold_cv(PCA_new, 8) # split the training data into 8
folds

reci <- recipe(train_y~.,data = PCA_new)  # the basic recipe we
use
```

### XGB model with 149 predictors and making predictions
```{r}
set.seed(666) # set seed for reproducing results

# xgboost model with our tuned hyper-parameters
xgb_predict_model <- boost_tree( trees = 1000, tree_depth = 8,
min_n = 5, loss_reduction = 5.638207e-10,
                                 sample_size = 0.2351550, mtry =
131,learn_rate = 0.004848720) %>% set_engine("xgboost") %>%
set_mode("regression")



xgb_fit <- xgb_predict_model %>% fit(train_y~., data = PCA_new) #
fit the model to our selected training data

pred <- test$id %>%
  bind_cols(predict(xgb_fit, new_data = test)) # making
predictions

colnames(pred) <- c("Id", "Predicted")

write_csv(pred, "xgb_pred") # storing the predicted data
```
~~~

~~~
### sort the predictors in the xgboost model by its importance
```{r}
train_matrix <- data.matrix(PCA_new)
mat <- xgb.importance (names(train_matrix),model = xgb_fit$fit) #
sort the predictors in the xgboost model by its importance
~~~

### create sgb model with the 80 most important predictors
```{r}

var_80 <- PCA_new[,colnames(PCA_new) %in% mat$Feature[1:80]] #
select the top 80 most important predictors

var_80 <- cbind(train_y,var_80) # bind them with our response
variable

reci <- recipe(train_y~.,data = var_80)  # the receipe we use in
this model

set.seed(666)

# xgboost model with our tuned hyper-parameters
xgb_predict_model_80 <- boost_tree(trees = 1000, tree_depth = 6,
min_n = 11, loss_reduction = 2.166431e-10,
                                sample_size = 0.3461942, mtry =
27,learn_rate = 0.021839515) %>% set_engine("xgboost") %>%
set_mode("regression")



xgb_fit_80 <- xgb_predict_model_80 %>% fit(train_y~., data =
var_80) # fit the model to our selected training data
```

~~~
pred_80 <- test$id %>%
  bind_cols(predict(xgb_fit_80, new_data = test))

colnames(pred_80) <- c("Id", "Predicted")

write_csv(pred_80, "xgb_pred_80")
```
~~~

Appendix: Team Member Contributions

Jingyi Lang was in charge of writing R codes to preprocess the training data set, creating different candidate models to make predictions for test data, and writing and proofreading the final report.

Zhitong Zhou was responsible for writing R codes for testing different hyperparameters and recipes in the candidate models, performing Principal Component Analysis for variables, creating XGBoost models to make predictions, and writing the final report.

Olivia Wang was in charge of writing R codes for preprocessing the training data set and variable selection methods, creating all candidate models to make predictions for test data, and proofreading the final report.

Yanjun Jiang was responsible for writing R codes to create all candidate models to make predictions, test the models with different recipes, and write the final report.

All four of us participated in writing R codes to create models and make predictions for the test data and the report's writing.