Group 9 Classification Report

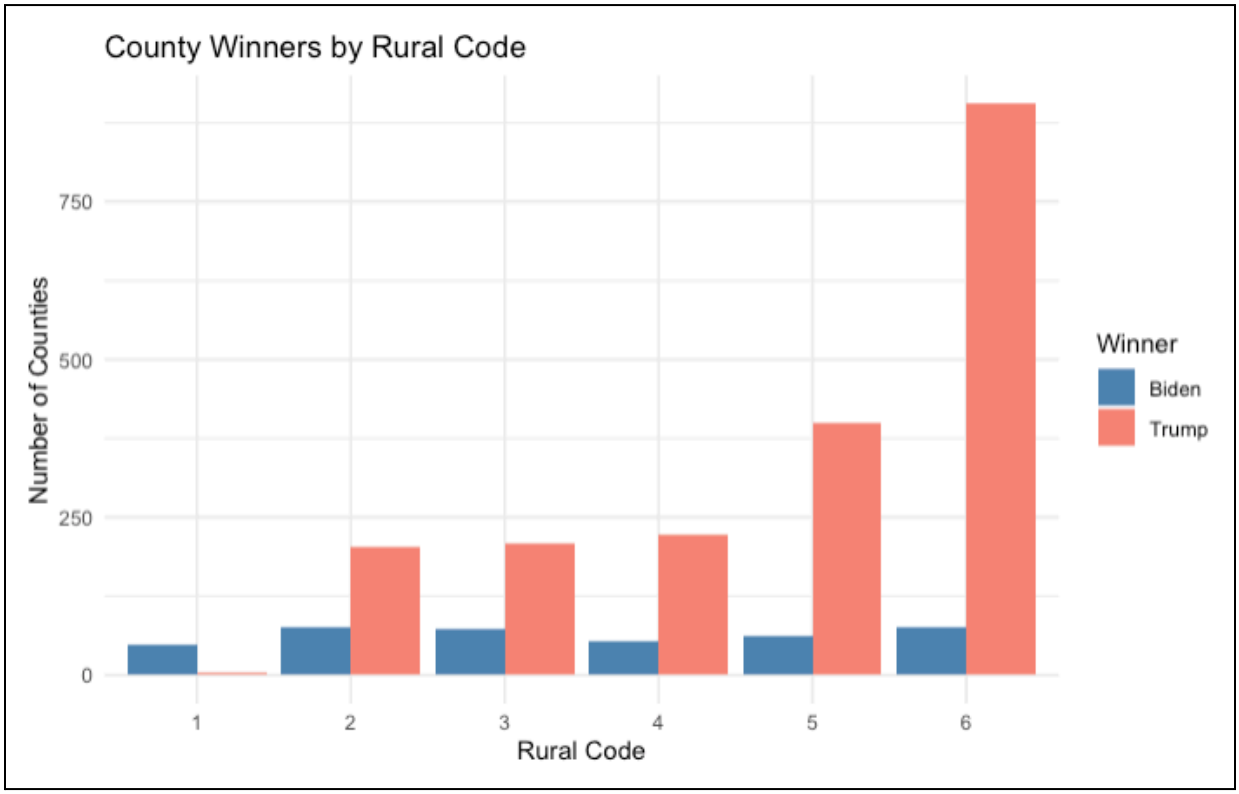
Introduction

Predicting electoral outcomes at the county level has become an important area of study in political science and data science, as it provides insights into the demographic, economic, and social factors that shape voting behavior. Prior research highlights the role of variables such as population density, education levels, racial composition, and economic indicators in influencing electoral outcomes (Smith & Rodriguez, 2022). For example, urban counties with higher levels of educational attainment have historically leaned toward Democratic candidates, while rural areas with lower population density often favor Republican candidates (Smith & Rodriguez, 2022). In this project, we use county-level data to model and predict the winner of the 2020 U.S. presidential election, examining which predictors are most strongly associated with political outcomes.[1]

[1] Smith, J., & Rodriguez, M. (2022). *County-level predictors of U.S. presidential election outcomes*. Journal of Political Data Science, 5(2), 115–134.

Exploratory Data Analysis

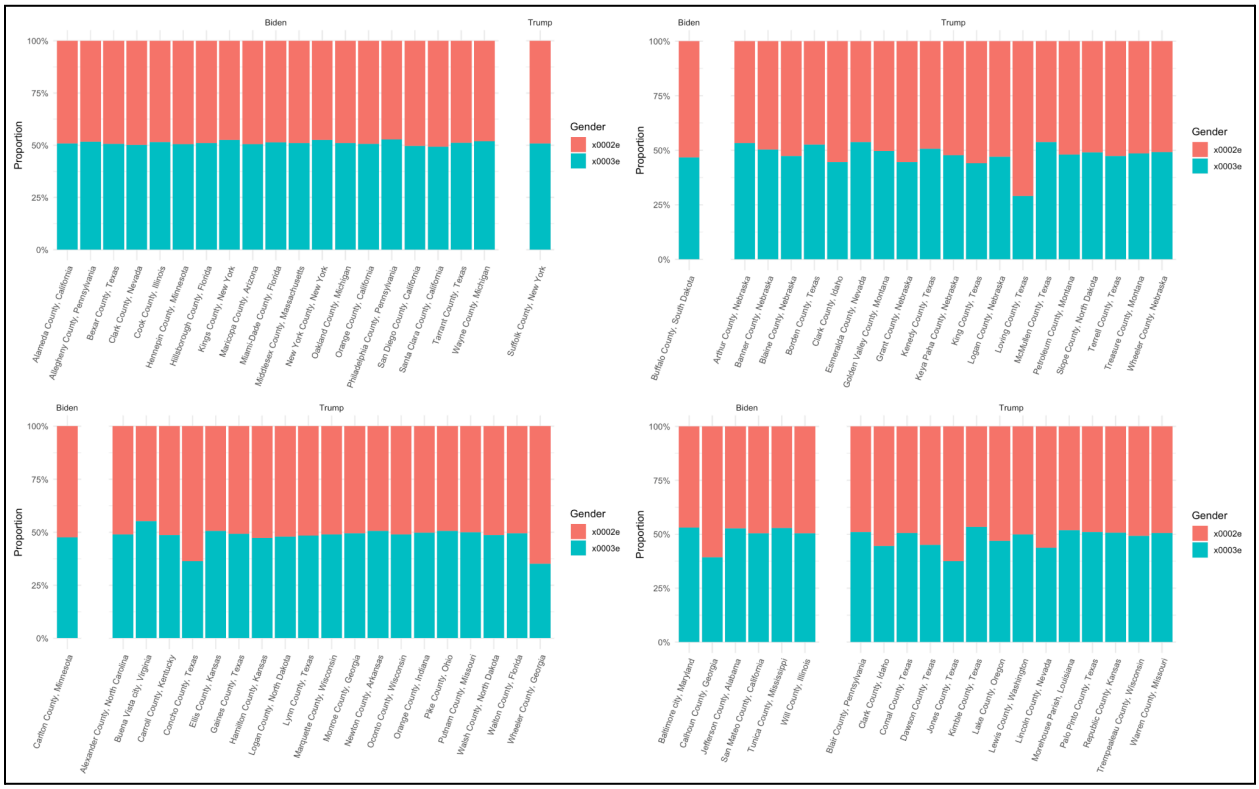
1. Bar Chart of County Winners by Rural Code



The bar chart above shows the number of counties that the two candidates won, split into rural/urban codes. According to the data description, 1 represents the most urban and 6 represents the most rural. As we can see, Trump had a higher number of county wins in counties that were more rural, whereas Biden had roughly an even split across all ranges. This suggests that the rural code will be a good categorical predictor and may improve the predictive performance of the model if included.

Another factor to consider is that this visualization demonstrates how Trump won many more counties compared to Biden. We are wary of this information, as it may bias the model with the disproportionate class occurrences. A model could achieve a high accuracy score while testing model performance by predicting Trump as the winner for most of the counties, without being truly accurate or generalizable to additional data.

2. Gender Demographics in Different Counties

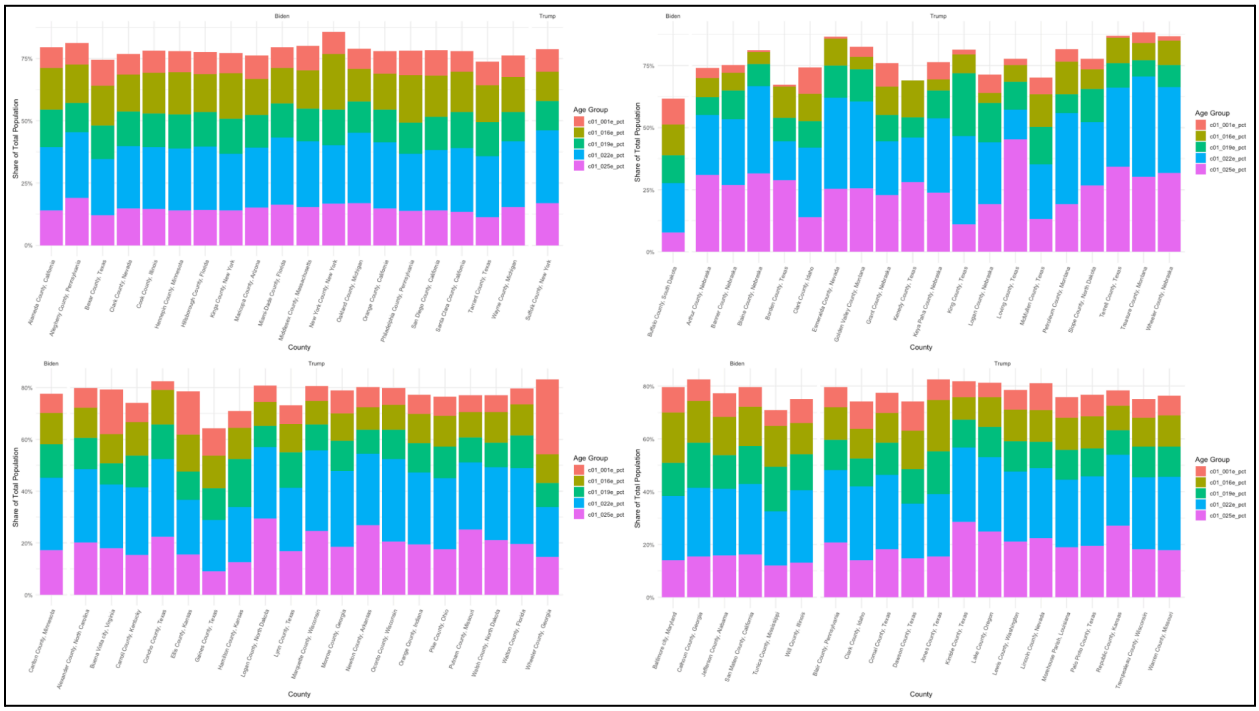


The chart above shows the gender demographics in various counties. The red and blue bars represent the proportion of males and females, respectively. The counties are also split according to the winning candidates, Biden (left) and Trump (right).

Top Left Plot: Gender population split for 20 counties with the highest population
Top Right Plot: Gender population split for 20 counties with the lowest population
Bottom Left Plot: Gender population split for 20 random counties (seed = 101)
Bottom Right Plot: Gender population split for 20 random counties (seed = 200)

A common presumption is that Trump won counties with a higher proportion of males. However, when we look at the various charts, we can see that this relationship is not conclusive. There are many counties that Trump and Biden won, with a relatively even split of male and female voters. It is true that there are some counties that Trump won that have a higher proportion of males. However, the same is true for some counties won by Biden, albeit less often. This suggests that the gender demographic information can provide some predictive power to the model, but it may be weaker than other predictors.

3. Age Demographics in Different Counties

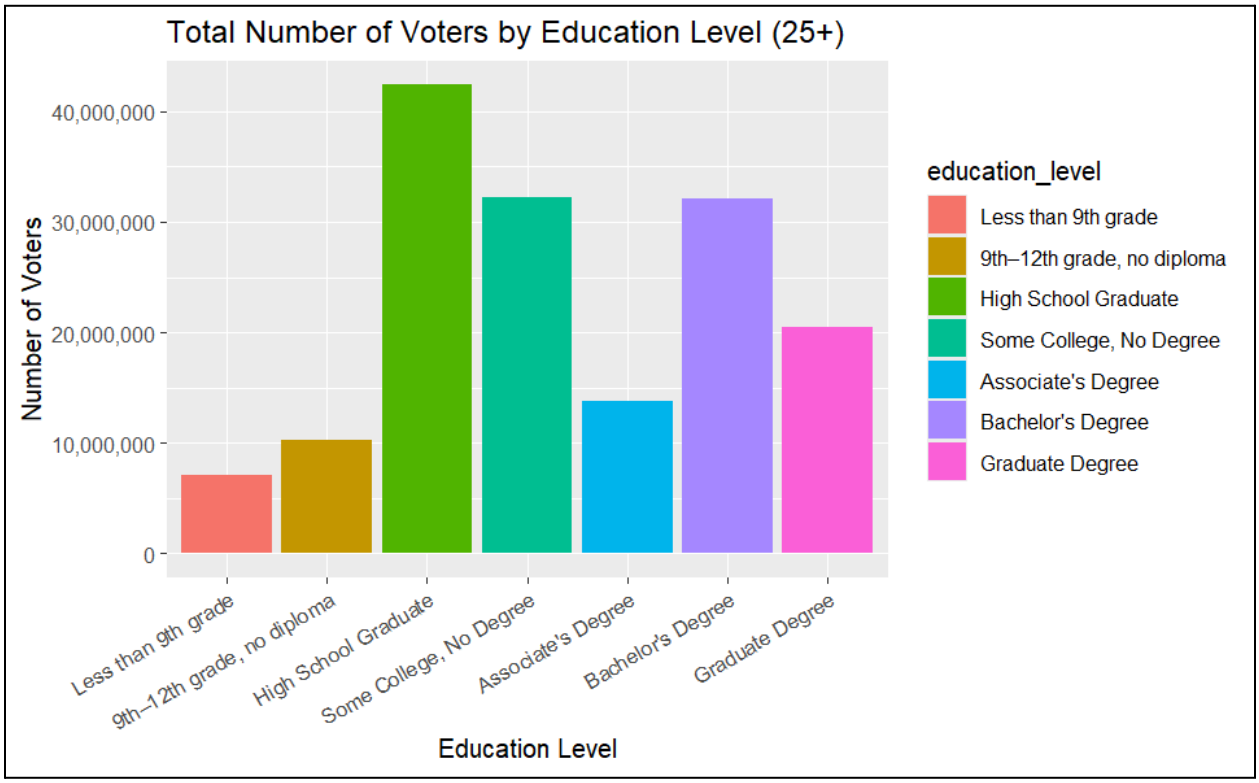


The chart above shows the age demographics in different counties. The bars do not reach 100% because the non-voting age group of ages 0-17 is not included in the plot. Red represents ages 18-24, yellow represents ages 25-34, green represents ages 34-44, blue represents ages 45-64, and purple represents ages 65 and above. The counties are also split according to the winning candidates, Biden (left) and Trump (right).

Top Left Plot: Age population split for 20 counties with the highest population
Top Right Plot: Age population split for 20 counties with the lowest population
Bottom Left Plot: Age population split for 20 random counties (seed = 101)
Bottom Right Plot: Age population split for 20 random counties (seed = 200)

Based on the charts, we can see that for the counties that Biden won, there is a larger proportion of young people (red, yellow, and green groups) compared to older people (blue and purple groups). In counties that Trump won, we see the opposite trend. Based on this visualization, we believe that age may be a strong predictor in determining the winner of a county. It may also prove useful to further bin the age groups into smaller bins (young vs. old).

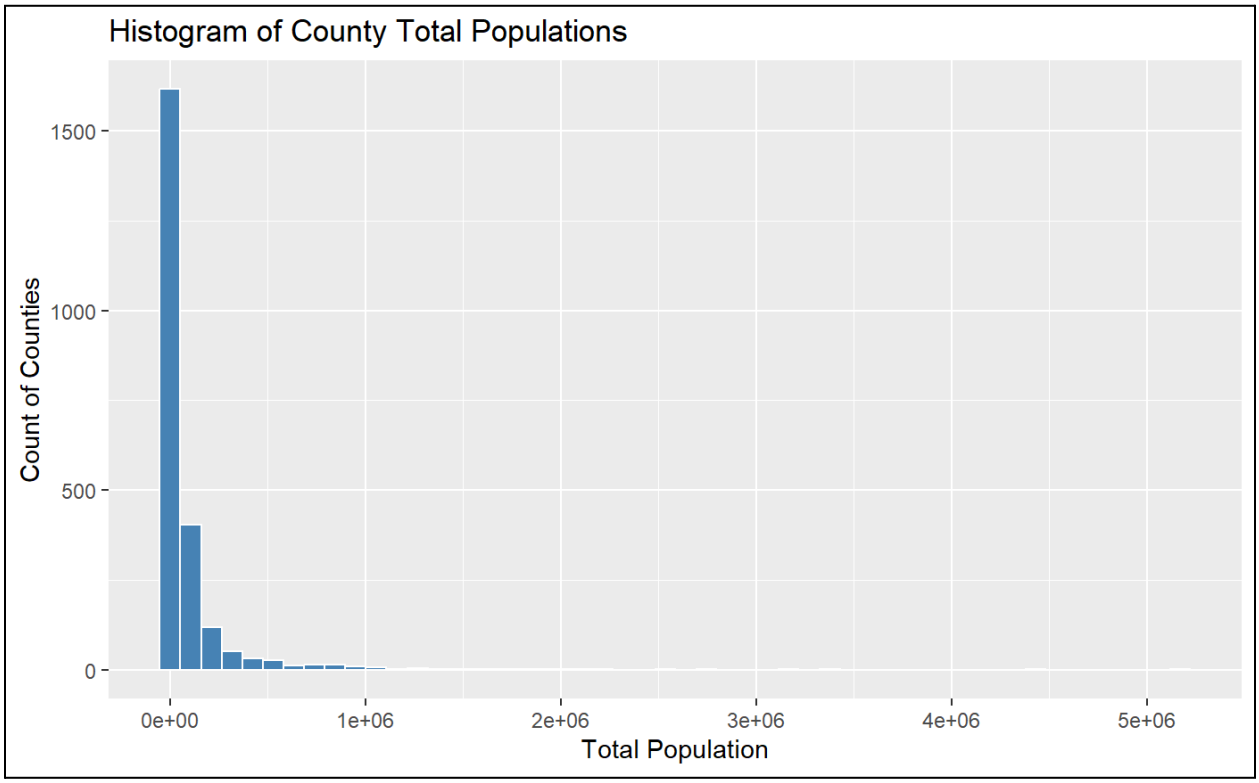
4. Total Number of Voters by Education Level (Age 25+)



The bar chart above illustrates the total number of voters aged 25 and older across various educational levels. There are clear differences in population size across categories. High school graduates comprise the largest group, followed by those with some college education or a bachelor's degree.

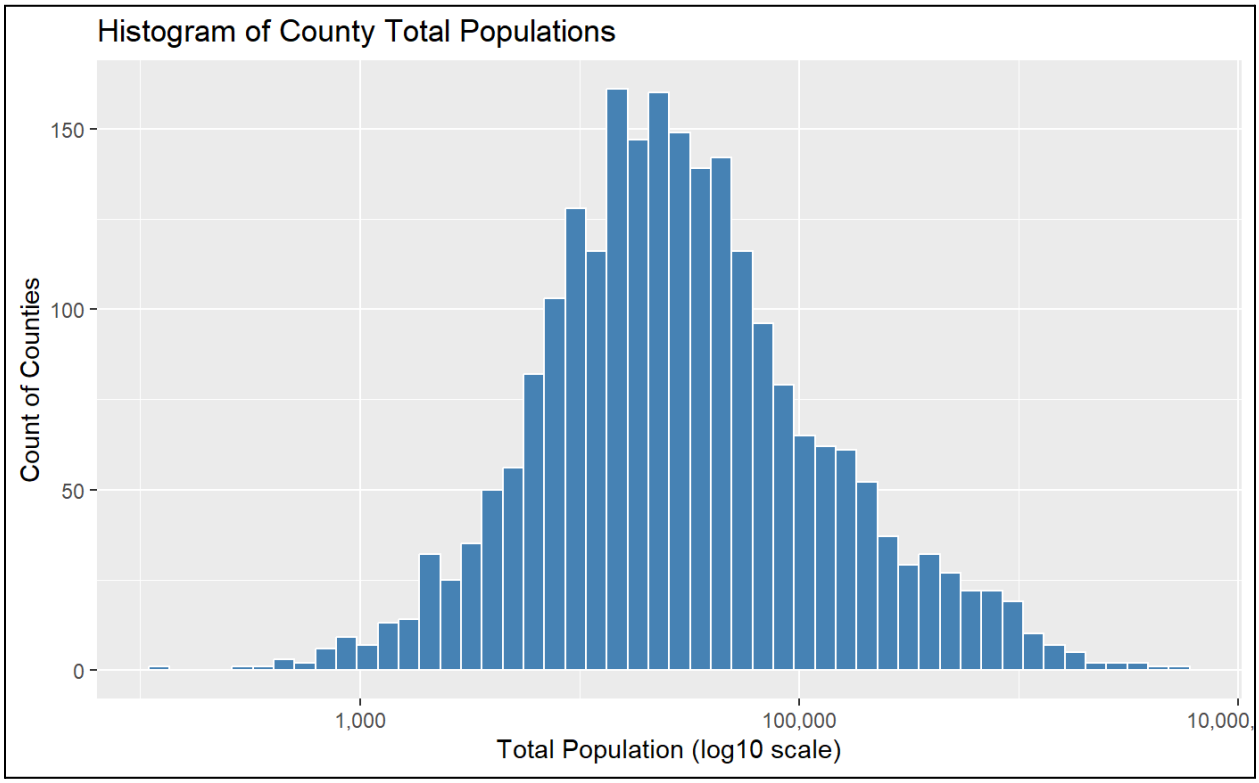
This visualization could inform us of potential transformations to this data to improve our model stability, such as through binning and grouping. Understanding which education levels are most common also informs us of potential interactions with demographic variables like age or income. Younger voters with higher education levels may vote differently from older voters with the same education. Education also often correlates with income.

5. Histogram of Total Population Counts



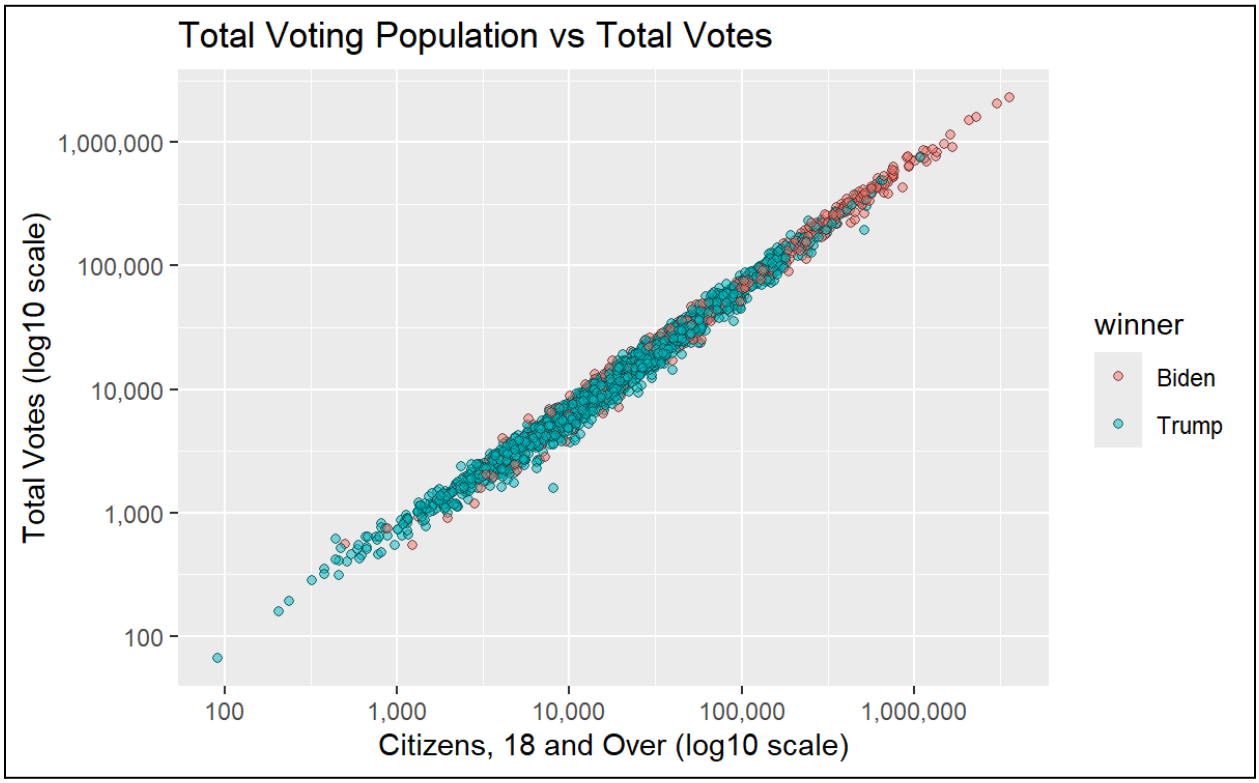
As part of our initial exploration, we plotted the distribution of `x0001e` (total population in a county) values. Because it was extremely right-skewed, we decided to perform a log-10 transformation on the variable.

6. Histogram of Total Population Counts, Log-transformed



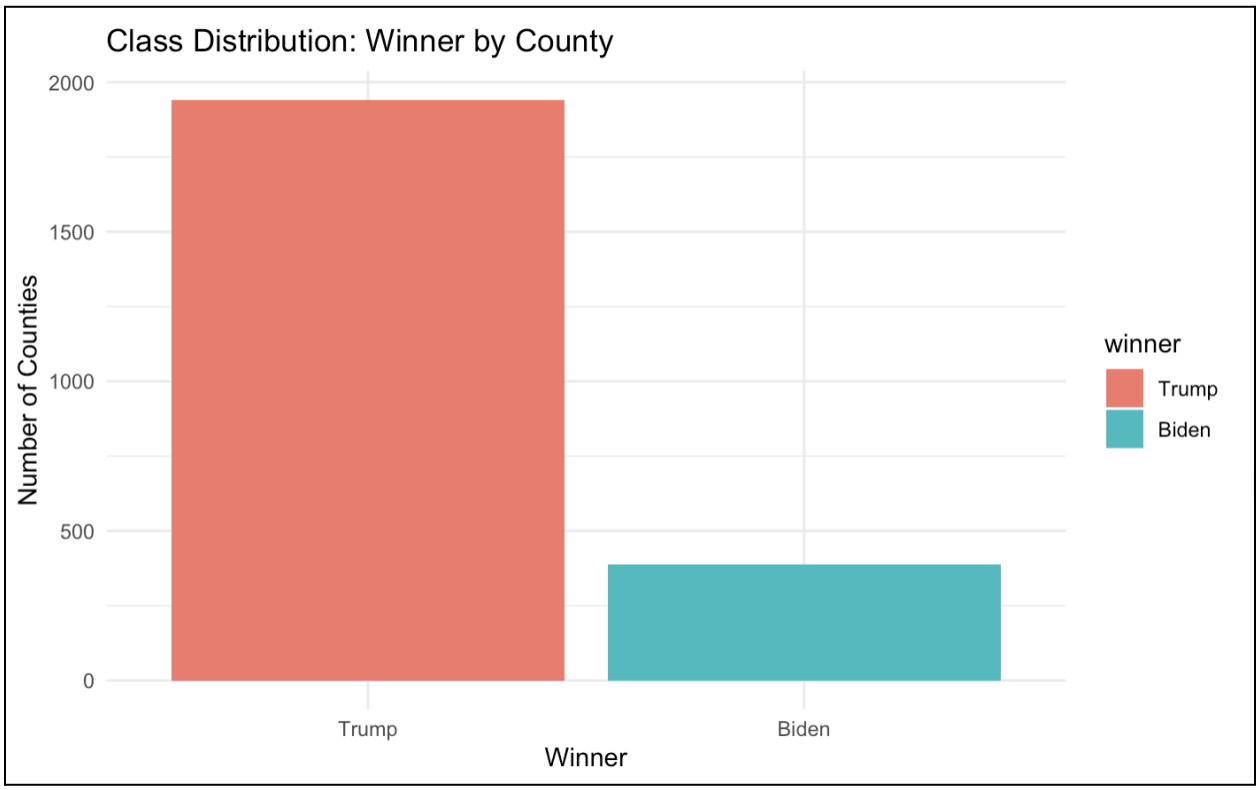
The above plot shows the `x0001e` (total population) variable after a log-transformation. With the data in a more normal, roughly bell-shaped distribution, we can tell that most counties fall between about 10,000 and 100,000 people, with far fewer at the extremes. The original skew shows that a few large counties hold a disproportionate share of the population, while the majority of counties are much smaller. To avoid these values over-leveraging our modeling and better compare counties, a log-transformed population is a more appropriate feature than the raw population. We came to similar conclusions in our exploration of the `x0086e` (total housing units), `income_per_cap_[year]` (income per capita for the county in a given year), and `gdp_[year]` (GDP for the county in a given year)

7. Scatterplot of Total Population vs Total Votes



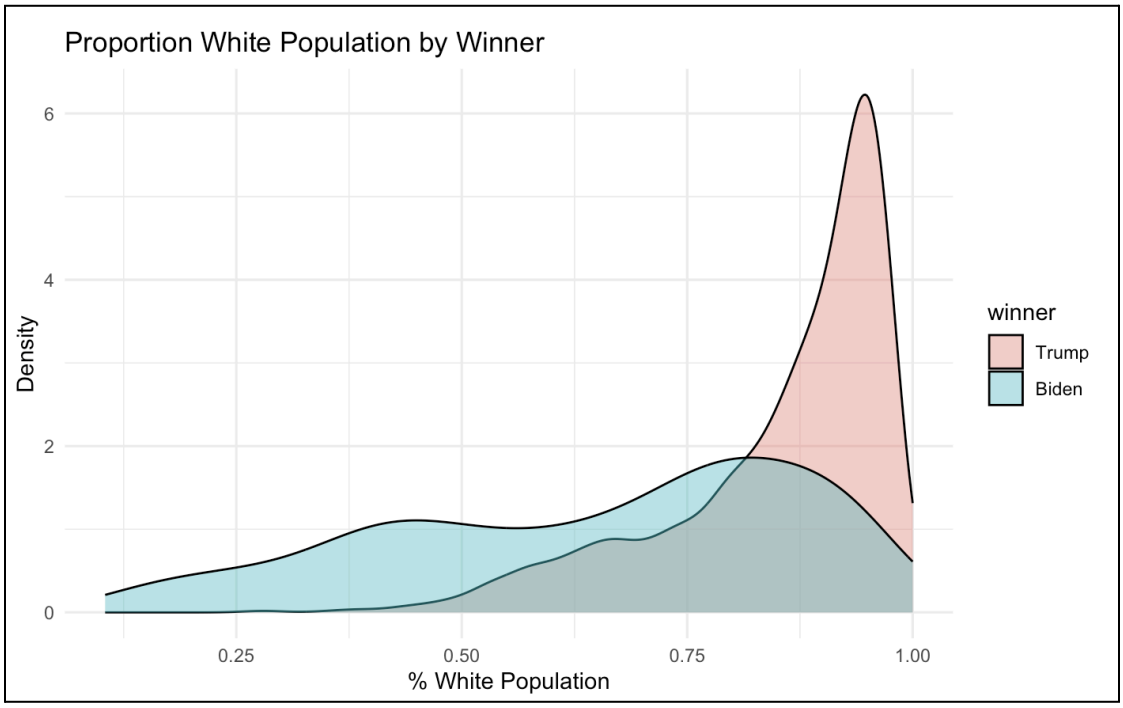
This scatterplot illustrates the relationship between `total_votes` and `x0087e`, the estimated population of voting-age citizens in the counties of the dataset. The relationship is very strongly linear, and the close clustering while being on a log-log scale indicates that the `total_votes` variable scales proportionally with the size of the voting-age population. The coloring by winner (Biden vs Trump) appears to show a slight separation in this relationship, with Biden tending to win counties of larger voting populations and Trump tending to win counties of smaller voting populations. This suggests that population size by itself may be a moderate predictor of the winning candidate of a county.

8. County-Level Outcome Distribution



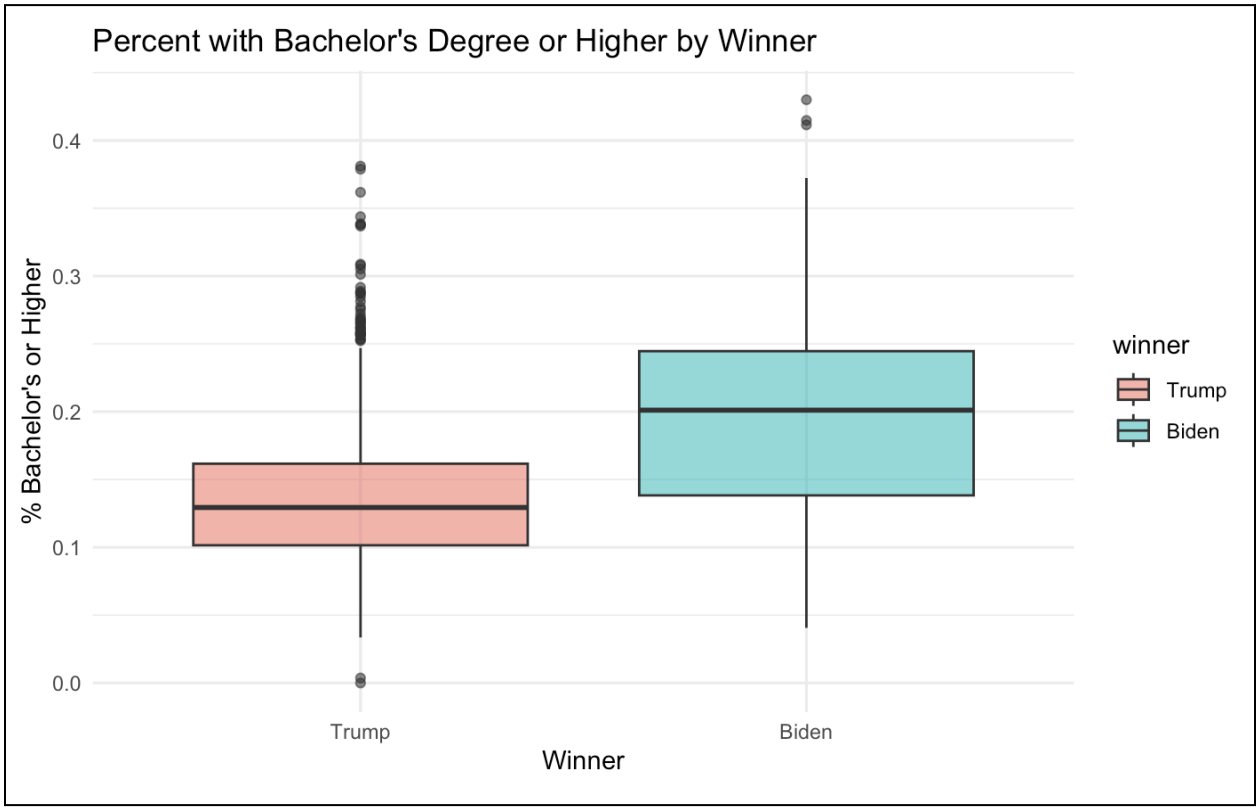
The bar chart shows the distribution of county-level winners. The majority of counties were won by Trump, while a smaller portion went to Biden. This highlights a well-known electoral pattern: although Biden won the popular vote and the election overall, Trump carried more counties, which tend to be more rural and less densely populated. This imbalance underscores the importance of considering population density and urban–rural divides as strong predictors of voting outcomes.

9. Proportion of White Population by Winner



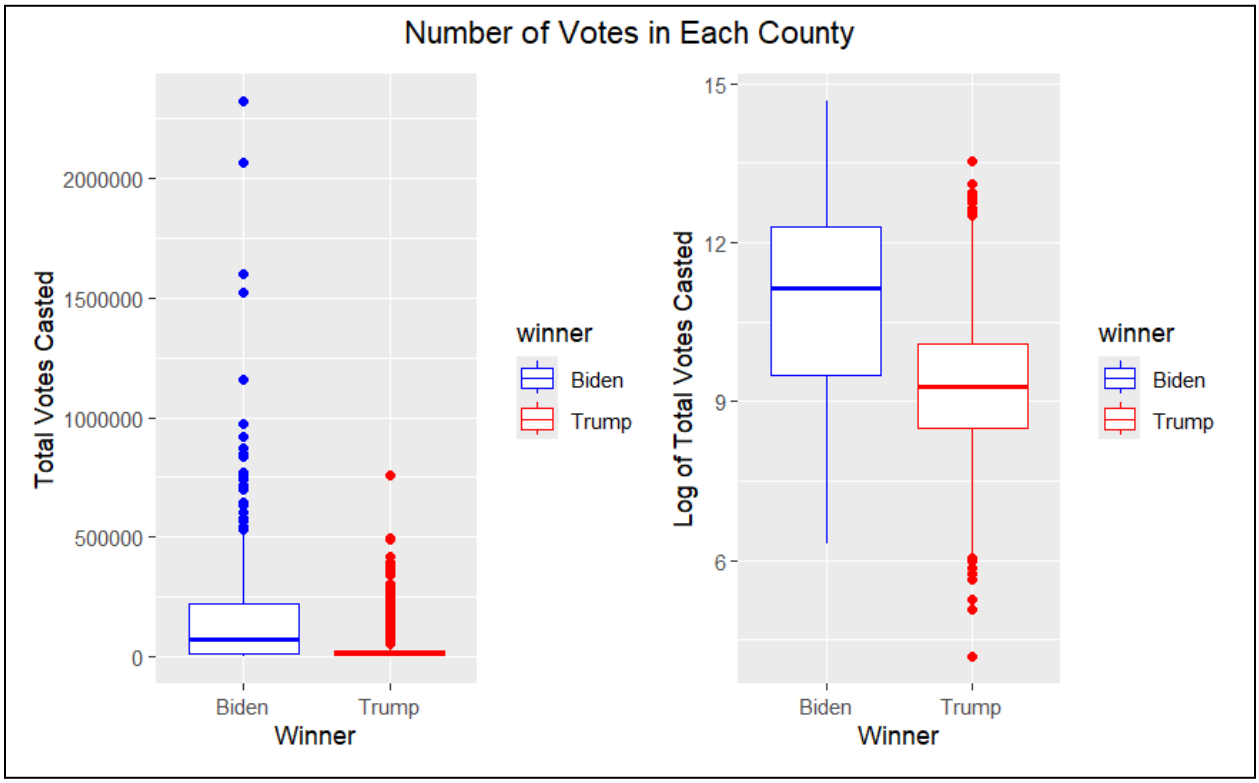
The density plot shows the distribution of counties' white population proportion by candidate. Counties with a higher percentage of white residents were more likely to vote for Trump, whereas Biden-leaning counties generally had lower proportions of white residents. This suggests that racial composition is a meaningful predictor of county-level election results, consistent with prior research linking demographic diversity to Democratic support in urban and suburban areas.

10. Education Level and Voting Outcomes



The boxplot compares the educational attainment of counties by winner. Biden counties generally show higher percentages of residents with a bachelor's degree or higher, while Trump counties cluster around lower education levels. This supports prior findings that education level is a strong predictor of Democratic support, especially in urban and suburban areas, while lower levels of education are associated with Republican support in rural counties.

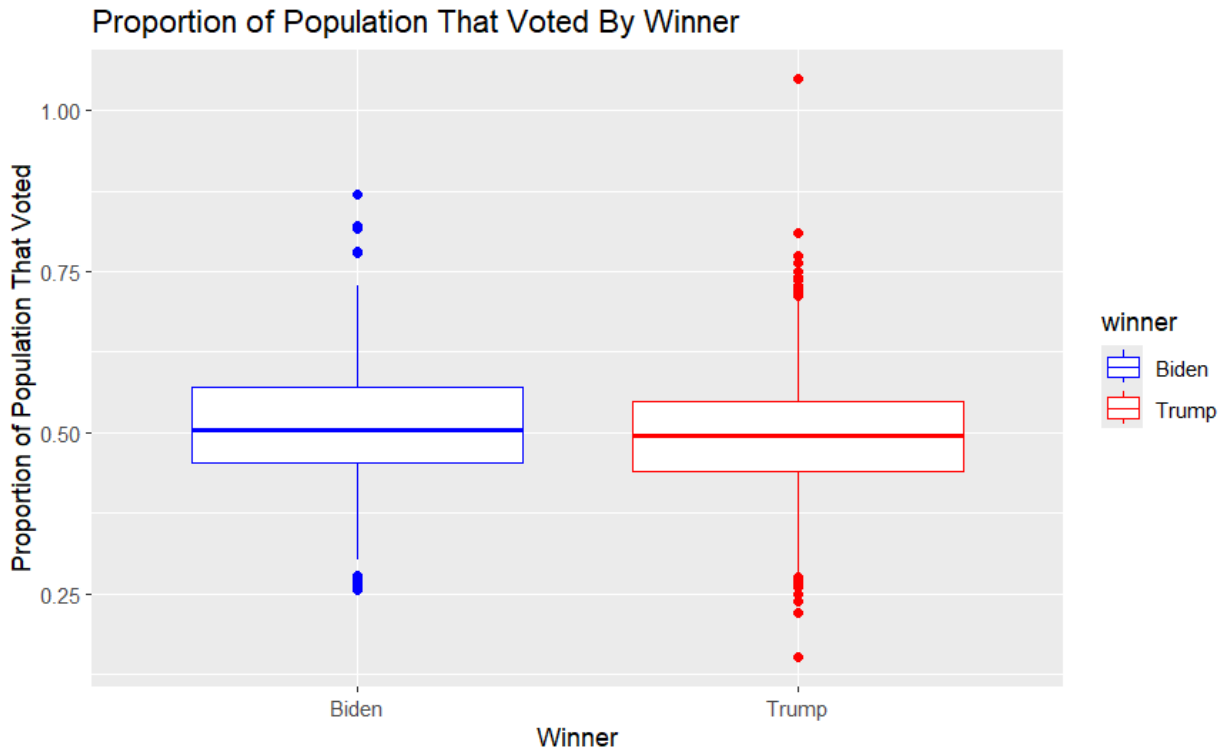
11. Boxplot of Number of Votes in Each County



This boxplot illustrates the total number of votes collected in each county and separated by who won in each county. This graph shows that counties that had more votes collected were more likely to have Biden as the winner than Trump, which can be better visualized by taking the natural log of the number of votes collected in each county.

Similar to the scatterplot earlier, this suggests that `total_votes` could be useful in predicting winning candidates. However, we are also interested in exploring other applications of the `total_votes` variable, such as voter turnout. While Biden often won in counties that had more votes, a county can only have as many votes as the total population in that county.

12. Boxplot of the Voter Turnout, By Winner



This boxplot follows up on the previous boxplot by showing the proportion of people who cast votes in each county. The counties that had Biden as their winner had a marginally higher voter turnout than the counties that had Trump as their winner. Due to this, we suspect that voter turnout is a relatively weak predictor. Combining the information from this graph with the information from the previous graph, it seems that `total_votes` is a better indicator of who won in the county than the proportion of people who voted in each county. This aligns with data from the election that showed that Biden won in counties that have higher populations, like Los Angeles and San Diego, and Trump won in counties that have lower populations.

Preprocessing / Recipes

Logistic Regression Recipe

```
glm_recipe <- # M2
  recipe(winner ~ ., data = train_data) |>
  step_mutate(
  across(x0002e:x0085e, ~ .x / x0001e),
  across(x0087e:c01_027e, ~ .x / x0001e)) |> # change demographic info to proportion
  step_log(x0001e, x0086e, income_per_cap_2016:gdp_2020) |> # log population, housing units, income info
  step_mutate(x2013_code = factor(x2013_code)) |>
  step_dummy(x2013_code) |> # turn rural information into dummy variable
  step_rm(id, total_votes, x0033e, x0029e, x0034e) |> # remove non-predictors + duplicates
  step_zv(all_predictors()) |>
  step_impute_median(income_per_cap_2016:gdp_2020) |>
  step_normalize(all_numeric_predictors()) |>
  step_rm(gdp_2016:gdp_2020, x0002e:x0003e)
```

step_mutate: For the logistic regression model, we converted most of the demographic variables into proportions/percentages by dividing these variables by `x0001e` (total population). This would allow us to avoid the absolute values from skewing the data, since certain variables would have larger demographic values, simply due to a larger population size. By converting the variables into proportions instead, we are able to make better comparisons between them. In addition, the second step_mutate step was used to convert the `x2013_code` (rural/urban code) into a factor, which would allow us to then make this variable categorical.

step_log: Variables which were not based on a proportion of the total population, such as `x0001e` (total population), `x0086e` (total housing units), `income_per_cap_[year]` (income per capita for the county in a given year), and `gdp_[year]` (GDP for the county in a given year) were right skewed when plotted on a histogram. To correct this skewed distribution, these variables were log-transformed, to prevent certain observations from heavily affecting the model.

step_dummy: `x2013_code` (rural/urban code) was converted into a dummy variable.

step_zv: As a precautionary measure, this step was added to remove any columns that have zero variance (only a single value).

step_impute _median: Due to 37 observations having missing values for the `income_per_cap_[year]` and `gdp_[year]` variables, the missing values were imputed with the median of the other observations that were available so that we can still generate the model.

step_rm: There are two step_rm steps in this recipe. The first step_rm removed the variables that are not predictors (`id`, `total_votes`), as well as variables that are duplicate columns (`x0029e`, `x0033e`, `x0034e`). The second step_rm removed all the variables that we felt were providing redundant information, due to other variables in the model already providing this information. `gdp_[year]` variables were removed since we felt that `income_per_cap_[year]` already provided similar information. `x0002e` (total population of males) and `x0003e` (total population of females) were also removed because there is already a variable for the total population of males and females over the age of 18 (over the voting age), thus we felt that including all of the variables would add redundancy.

step_normalize: All numeric variables were then normalized. This is particularly important for penalized logistic regression, since the original values (counts of people within a demographic) would distort the distances and punish certain variables more than others.

Random Forest Recipe

```
rf_recipe <- # M1
  recipe(winner ~ ., data = train_data) |>
  step_mutate(
  across(x0002e:x0085e, ~ .x / x0001e),
  across(x0087e:c01_027e, ~ .x / x0001e)) |> # change demographic info to proportion
  step_log(x0001e, x0086e, income_per_cap_2016:gdp_2020) |> # log population, housing units, income info
  step_mutate(x2013_code = factor(x2013_code)) |>
  step_dummy(x2013_code) |> # turn rural information into dummy variable
  step_rm(id, total_votes, x0033e, x0029e, x0034e) |> # remove non-predictors + duplicates
  step_zv(all_predictors()) |>
  step_impute_median(income_per_cap_2016:gdp_2020) |>
  step_rm(gdp_2016:gdp_2020, x0002e:x0003e)
```

The random forest recipe is almost the same as the logistic regression recipe. The only difference is the removal of step_normalize. Random Forest does not require the variables to be normalized, since the model can work with the original values.

Alternative Random Forest Recipe

```
rf_recipe <-
  recipe(winner ~ ., data = train_data) |>
  step_mutate(turnout = total_votes / x0087e) |> # add voter turnout predictor
  step_mutate(
  |  across(x0002e:x0085e, ~ .x / x0001e),
  |  across(x0087e:c01_027e, ~ .x / x0001e)) |> # change demographic info to proportion
  step_log(x0001e, x0086e, income_per_cap_2016:gdp_2020) |> # log population, housing units, income info
  step_mutate(x2013_code = factor(x2013_code)) |>
  step_dummy(x2013_code) |> # turn rural information into dummy variable
  step_rm(id, total_votes, x0033e, x0029e, x0034e) |> # remove non-predictors + duplicates
  step_zv(all_predictors()) |> # remove predictors with zero variance
  step_impute_median(income_per_cap_2016:gdp_2020) |>
  step_discretize(income_per_cap_2016:gdp_2020, options = list(cuts = 5)) |> # sort income and gdp info
into bins
  step_corr(all_numeric_predictors(), threshold = 0.9) # remove highly correlated predictors
```

We also tested alternative random forest recipes with different steps. Predictions made using a random forest model that employed the recipe shown above were submitted to Kaggle. The resulting private score was tied for highest accuracy with the private score of a random forest model based on the original recipe, with added tuning. However, we did not ultimately employ this recipe in further modeling due to time constraints. Ideally, we would have explored more pre-processing steps to try to enhance the accuracy of our random forest modeling. A handful of steps were changed in the original recipe:

step_mutate: Another predictor was added to represent voter turnout, as given by dividing `total_votes` by the voting population `x0087e` (Citizen, 18 and over population).

step_discretize: Predictors containing information for income per capita and GDP were sorted into bins to produce more interpretable model features.

step_corr: Instead of removing the GDP information and the total populations of males and females, we included a step to remove highly correlated predictors. After this step, there were a remainder of 93 predictors. All income and GDP variables were retained.

Boosted Tree Recipe (xgboost)

```
xgb_recipe <- # M1
  recipe(winner ~ ., data = train_data) |>
  step_mutate(
  across(x0002e:x0085e, ~ .x / x0001e),
  across(x0087e:c01_027e, ~ .x / x0001e)) |> # change demographic info to proportion
  step_log(x0001e, x0086e, income_per_cap_2016:gdp_2020) |> # log population, housing units, income info
  step_mutate(x2013_code = factor(x2013_code)) |>
  step_dummy(x2013_code) |> # turn rural information into dummy variable
  step_rm(id, total_votes, x0033e, x0029e, x0034e) |> # remove non-predictors + duplicates
  step_zv(all_predictors()) |>
  step_impute_median(income_per_cap_2016:gdp_2020) |>
  step_rm(gdp_2016:gdp_2020, x0002e:x0003e)
```

Same recipe as the Random Forest recipe.

K Nearest Neighbors Recipe

```
knn_recipe <- # M1
  recipe(winner ~ ., data = train_data) |>
  step_mutate(
  across(x0002e:x0085e, ~ .x / x0001e),
  across(x0087e:c01_027e, ~ .x / x0001e)) |> # change demographic info to proportion
  step_log(x0001e, x0086e, income_per_cap_2016:gdp_2020) |> # log population, housing units, income info
  step_mutate(x2013_code = factor(x2013_code)) |>
  step_dummy(x2013_code) |> # turn rural information into dummy variable
  step_rm(id, total_votes, x0033e, x0029e, x0034e) |> # remove non-predictors + duplicates
  step_zv(all_predictors()) |>
  step_impute_median(income_per_cap_2016:gdp_2020) |>
  step_normalize(all_numeric_predictors()) |>
  step_rm(gdp_2016:gdp_2020, x0002e:x0003e) |>
  step_pca(all_numeric_predictors(), num_comp = tune())
```

Most of this recipe is the same as the logistic regression recipe. We included step_normalize since distances are important for the KNN model, and the original magnitudes can skew results.

We also included step_pca since KNN can often fall into the trap of the curse of dimensionality when there are too many variables present. Thus, we added PCA as a way of reducing the dimensions and variance, in the hopes of improving model accuracy. This value is tuned during the processing step.

Candidate Models

We created various models to predict the `winner` of each county in the test set. The following 5 models are the main models that were created:

1. glm_M2_mixture (Penalized Logistic Regression)
    a. We attempted a logistic regression model that was tuned based on the penalty and the mixture (0 = pure ridge, 0.5 = net-elastic, and 1 = lasso). This would act as a base model to base our performance on.
2. rf_M1 (Random Forest)
    a. Random Forest was one of the strongest-performing models. In fact, after the private score on Kaggle was released, one iteration of this model (rf_M1.1), which had more tuning and tied for the highest accuracy out of all our other models that were submitted. We kept the hyperparameter `trees` at a constant value to reduce the number of hyperparameters that needed tuning so that the grid search could be more precise.
3. xgb_M1_tunebayes2 (Boosted Tree)
    a. A boosted tree model was attempted with the `xgboost` engine. After a random grid search, the model was further tuned using an iterative search. After two rounds of tune_bayes, it gave strong results. This model was selected for final submission in the Kaggle competition and produced our team's highest public score.
4. knn_M1 (K Nearest Neighbors)
    a. We also tried using a KNN model, but ended up having poorer performance than our tree models. The accuracy was also weaker than the logistic regression model.
5. stack_M1 (Stacked Model)
    a. This model is a stacked model that combines all of the previously created models together. The exception is the xgb_M1_tunebayes2 model. Due to time constraints, the model that was added to the stack was the xgb_M1 (which did not have the additional iterative searches). This may have impacted the performance of the stack model. Overall, the stack model was one of our highest-performing models and was also selected for the final submission in the Kaggle competition.

Attempted Candidate Models

Model Identifier	Type of Model	Engine	Recipe Used	Hyperparameters
glm_M2_mixture	Penalized Logistic Regression	glmnet	glm_recipe	`penalty`:0.003290345 `mixture`: 0.50
rf_M1	Random Forest	ranger	rf_recipe	`trees`: 1000 `mtry`: 30 `min_n`: 6
xgb_M1_tunebayes2	Boosted Tree	xgboot	xgb_recipe	`trees`: 1000 `mtry`: 30 `min_n`: 3 `tree_depth`: 3 `learn_rate`: 1.061884 `loss_reduction`: 2.007367 `sample_size`: 0.9611351
knn_M1	K Nearest Neighbors	kknn	knn_recipe	`neighbors`: 10 `num_comp`: 20
stack_M1	Stacked	Multiple	Multiple	`penalty`: 0.01 `mixture`: 1

Model Evaluation and Tuning

Our models' performances were evaluated based on their overall accuracy after a 10-fold cross-validation. Other metrics could have been chosen for model evaluation, such as ROC AUC or Balanced accuracy. However, since the competition only focuses on pure accuracy, we chose this metric. All models were tested using the same fold (that was created using seed 202). Each model was tuned, with different grid types being used depending on the model, since different engines had differences in how computationally expensive they were.

For example, the xgboost model is computationally expensive, so we chose to tune using a small random grid and use `tune_bayes` to potentially improve the model using iterative search. In our case, the iterative search actually improved our model quite significantly. For many of the tree models, the tree hyperparameter was set at 1000 to reduce the number of hyperparameters that needed tuning. This is to allow for more precise tuning of the other hyperparameters that may have a stronger impact on the performance of the model.

Different models also had different hyperparameters that required tuning. The following hyperparameters and ranges that they were tuned on are listed below:

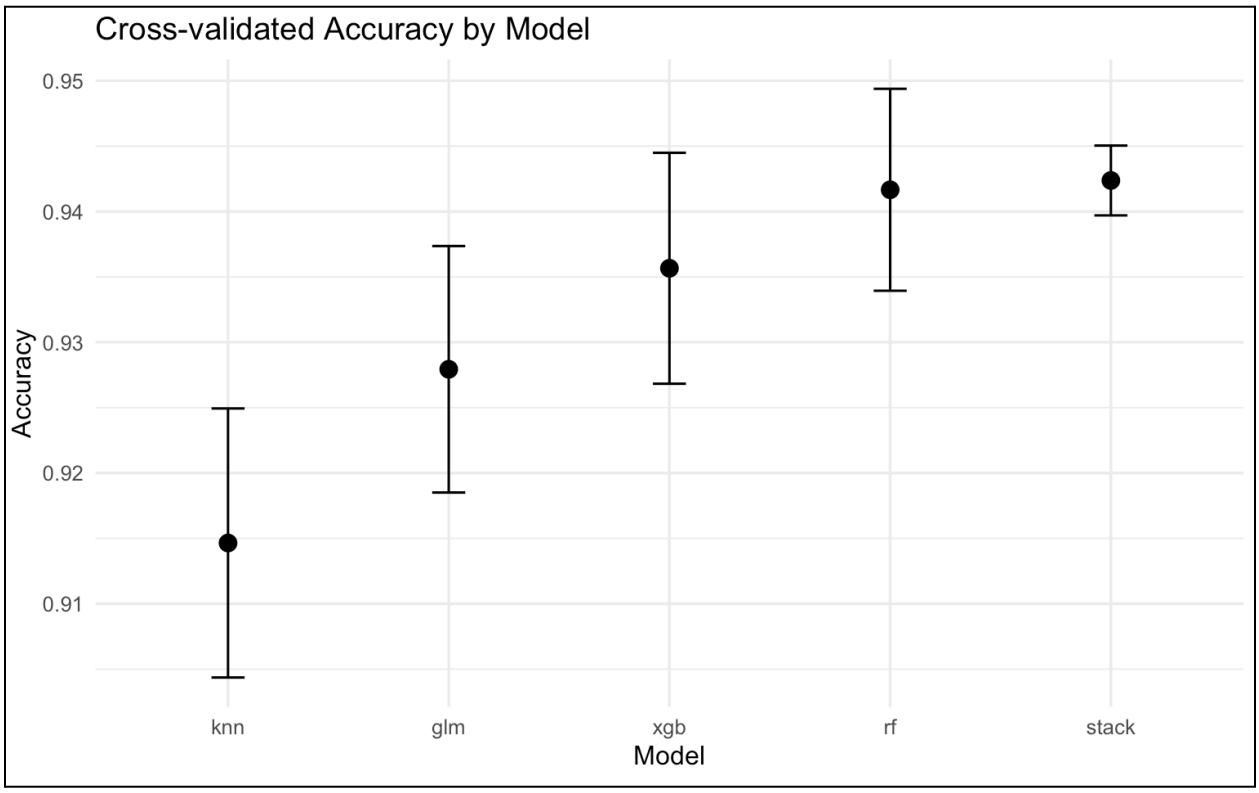
Tuning Details for Each Model

1. glm_M2_mixture (Penalized Logistic Regression)
    a. Tuned on a regular grid with 30 levels for penalty and 5 levels for mixture
    b. `penalty` was tuned from a range of 0.0001 to 0
    c. `mixture` was tuned from a range of 0 to 1
2. rf_M1 (Random Forest)
    a. Tuned on a regular grid with 5 levels
    b. `trees` was set at 1000
    c. `mtry` was tuned from a range based on the proportion of the predictors
    d. `min_n` was tuned from a range of 2 to 20
3. xgb_M1_tunebayes2 (Boosted Tree)
    a. Tuned on a latin hypercube grid with a size of 30
    b. `trees` was set at 1000
    c. `mtry` was tuned from a range based on the proportion of the predictors
    d. `tree_depth` was tuned from a range of 2 to 9
    e. `learn_rate` was tuned from a range of 0.003 to 0.2
    f. `loss_reduction` was tuned from a range of 0 to 8
    g. `min_n` was tuned from a range of 1 to 40
    h. `sample_size` was tuned from a range of 0.6 to 1.0
4. knn_M1 (K Nearest Neighbors)
    a. Tuned on a grid created using the crossing function
    b. `neighbors` was tuned from a range of 5 to 75
    c. `num_comp` was tuned from a range of 20 to 60
5. stack_M1 (Stacked Model)
    a. No tuning was required. The penalty is automatically tested.

Model Performance

Model Identifier	Metric Score (Accuracy)	SE of Metric
glm_M2_mixture	0.9279316	0.004810150
rf_M1	0.9416618	0.003940544
xgb_M1_tunebayes2	0.9356587	0.004506180
knn_M1	0.9146451	0.005250049
stack_M1	0.9423724	0.001361763

Model Comparison



Based on the model performance metric (accuracy) and the use of autoplot to compare the models, we determined that the best models were the stack model (stack_M1), random forest model (rf_M1), and boosted tree model (xgb_M1_tunebayes2). The model with the highest accuracy was the random forest model, with an accuracy score of 0.9416618. We noted that the stack model's standard errors were much smaller compared to the other models, potentially making it more robust in its predictions. However, since the boosted tree model and stack model both had public scores that were higher than the random forest model, we chose to submit those as our final models instead. With the boosted tree model producing the best

private score out of our submitted models, we selected it as our final model.

Discussion of Final Model

Final Model Selection

Our final model was the boosted tree model (`xgb_M1_tunebayes2`). While the random forest and stacked models also performed strongly, the boosted tree model produced the highest public score in our competition, making it our final model. The boosted tree model was iteratively refined using two rounds of Bayesian tuning (via `tune_bayes`), which allowed us to efficiently explore the hyperparameter space beyond an initial random grid search.

Strengths of the Model

One of the primary strengths of our approach was careful preprocessing of the county-level demographic and economic data. We accounted for the fact that there are significantly more Trump-winning counties than Biden-winning ones, which could otherwise bias the model toward the majority class. By converting most demographic variables into proportions rather than using absolute counts, we reduced the risk of larger counties dominating the model purely due to population size.

The use of Bayesian optimization twice was also a key strength that allowed our model to perform well. This enabled the boosted tree model to converge on strong hyperparameter values that may not have been identified through a single grid or random search. We also deliberately removed redundant variables, such as GDP, which was highly correlated with income per capita, and overlapping population counts, to reduce noise and avoid overfitting.

The boosted tree model itself provided flexibility in capturing nonlinear relationships and interactions between demographic and economic predictors, which made it well-suited for the complexity of county-level election prediction.

Weaknesses of the Model

A weakness of our boosted tree model is its higher standard error compared to our stacked model, indicating that its predictions may be more sensitive to sampling variability. Although we removed some redundant predictors, the dataset still may have contained correlated variables, which may have introduced multicollinearity and reduced interpretability. Tree-based models like xgboost are also prone to complexity, which can make them less transparent compared to simpler approaches like logistic regression.

Possible Improvements

We could have incorporated post-processing adjustments, such as applying thresholds to predictions based on county population. Errors in small rural counties, for example, may be less consequential than misclassifying large urban ones, so adjusting for population size could improve the relevance of results.

We could also have engineered our features further. Creating new variables through binning or grouping, such as splitting counties into population brackets or categorizing income levels, may help the model capture structural differences more effectively. Variable mutation may also have improved our model's interpretability.

Appendix: Final Annotated Script

Script for Final Model (xgb_M1_tunebayes2)

```
library(tidyverse)
library(tidymodels)

# Load data
test_data <- read.csv("test_class.csv")
train_data <- read.csv("train_class.csv")
train_data <- train_data |>
  select(-name) |>
  mutate(winner = factor(winner, levels = c("Trump", "Biden")))
# convert winner to factor before recipe step

# Create folds
set.seed(202)
cv_folds <- vfold_cv(train_data, v = 10, strata = winner)

# 1. Recipes (Pre-processing)

xgb_recipe <-
  recipe(winner ~ ., data = train_data) |>
  step_mutate(
  across(x0002e:x0085e, ~ .x / x0001e),
  across(x0087e:c01_027e, ~ .x / x0001e)) |>
# change demographic info to proportion
  step_log(x0001e, x0086e, income_per_cap_2016:gdp_2020) |>
# log population, housing units, income info
  step_mutate(x2013_code = factor(x2013_code)) |>
  step_dummy(x2013_code) |>
# turn rural information into dummy variable
  step_rm(id, total_votes, x0033e, x0029e, x0034e) |>
# remove non-predictors + duplicates
  step_zv(all_predictors()) |>
  step_impute_median(income_per_cap_2016:gdp_2020) |>
# impute missing values
  step_rm(gdp_2016:gdp_2020, x0002e:x0003e)
# remove redundant variables

xgb_prep <- prep(xgb_recipe, training = train_data) # Check changes
View(bake(xgb_prep, new_data = NULL))
```

```
# 2. Create model
xgb_spec <- boost_tree(
  trees = 1000, # keep trees constant
  learn_rate = tune(),
  tree_depth = tune(),
  min_n = tune(),
  loss_reduction = tune(),
  sample_size = tune(),
  mtry = tune()
) |>
  set_mode("classification") |>
  set_engine("xgboost",
             objective = "binary:logistic",
             eval_metric = "logloss",
             scale_pos_weight = (sum(train_data$winner == "Trump") /
                                   sum(train_data$winner == "Biden")))
# Since there are many more small counties that Trump won compared to
Biden, we want to punish an incorrect Biden prediction more than Trump
using scale_pos_weight

# 3. Create workflow
xgb_workflow <- workflow() |>
  add_model(xgb_spec) |>
  add_recipe(xgb_recipe)

# 4. Tuning grid
p <- ncol(bake(xgb_prep, new_data = NULL))
low <- max(1, floor(sqrt(p) / 2))
high <- max(low + 1, floor(p / 3))
# Make mtry range a proportion of the number of predictors

xgb_param_set <- parameters(
  learn_rate(range = c(0.003, 0.2)),
  tree_depth(range = c(2, 9)),
  min_n(range = c(1, 40)),
  loss_reduction(range = c(0, 8)),
  sample_size = sample_prop(range = c(0.6, 1.0)),
  mtry(range = c(floor(0.1*p), floor(0.5*p)))
)

set.seed(202)
```

```
xgb_grid <- grid_latin_hypercube(xgb_param_set, size = 30)

ctrl <- control_grid(
  save_pred      = TRUE,
  save_workflow  = TRUE,
  verbose        = TRUE)

my_metrics <- metric_set(accuracy, roc_auc)

xgb_res <-
  tune_grid(xgb_workflow,
            resamples = cv_folds,
            grid = xgb_grid,
            metrics = my_metrics,
            control = ctrl)


# 5. Checking best parameters
collect_metrics(xgb_res)
show_best(xgb_res, metric = "accuracy")

# 5a. Improve model with bayes_tune()

set.seed(202)
xgb_bayes <- tune_bayes(
  xgb_workflow,
  resamples = cv_folds,
  initial   = xgb_res,
  param_info = xgb_param_set,
  iter      = 20,
  metrics   = metric_set(accuracy, roc_auc),
  control   = control_bayes(verbose = TRUE,
                            save_pred = TRUE,
                            save_workflow = TRUE))


#Select best parameters
show_best(xgb_bayes)

# Continue tune_bayes one more time!
set.seed(202)
```

```
xgb_bayes2 <- tune_bayes(
  xgb_workflow,
  resamples  = cv_folds,
  initial    = xgb_bayes,
  param_info = xgb_param_set,
  iter       = 30,
  metrics    = metric_set(accuracy, roc_auc),
  control    = control_bayes(verbose = TRUE,
                             save_pred = TRUE,
                             save_workflow = TRUE,
                             no_improve = 15))
# Stop running if there is no improvement after 15 iterations

show_best(xgb_bayes2)
best_params <- select_best(xgb_bayes2, metric = "accuracy")
# Select the best parameters after tuning

# 6. Finalize workflow
final_wf <- finalize_workflow(xgb_workflow, best_params)

# 7. Fit on full training
final_fit <- fit(final_wf, data = train_data)

# 8. Predict on test set
results <- cbind(test_data |> select(id), final_fit |>
predict(test_data))
results <- rename(results, winner = .pred_class)

# 9. Create submission file
write_csv(results, "xgb_M1_tunebayes2.csv")
```

Appendix: Team Member Contributions

John Tan

- Submitted 6 models to Kaggle and had the best-performing model overall
- Created 10 different models and iterations
- Wrote Preprocessing/Recipes, Candidate Models section, and Appendix (Final Annotated Script)
- Created data visualizations 1, 2, 3 for Exploratory Data Analysis

Jillian Dessing

- Created 8 different models, submitted the best two to Kaggle
- Wrote Introduction
- Created data visualizations 8, 9, and 10
- Revised final report

Phoebe Chen

- Submitted 2 models to Kaggle
- Created data visualizations 5, 6, 7 for Exploratory Data Analysis
- Tested additional random forest recipe and documented in Preprocessing/Recipes
- Edited final report and its formatting

Joseph Lukas

- Submitted the first 7 models to Kaggle and submitted 9 models overall
- Created data visualizations 11 and 12 for Exploratory Data Analysis
- Reformatted graphs on final report

Saranna Lay

- Submitted 15 models to Kaggle
- Wrote Discussion of Final Model
- Created data visualization 4 for Exploratory Data Analysis