# Clustering Part 1

## Homework

### Prof. Bisbee

### Due Date: 2024-04-02

The last few times we have been talking about using a regression to characterize a relationship between variables. As we saw, by making a few assumptions, we can use either linear or logistic regression to summarize not only how variables are related to one another, but also how to use that relationship to predict out-of-sample (or future) observations.

We are now going to introduce a different algorithm - the `kmeans` algorithm. This specific algorithm is part of a larger class of algorithms/models that have been designed to identify relationships within the data. This is sometimes called “clustering” or “segmentation”. The goal is to figure out clusters/segments of data that are “similar” based on observable features.

The goal is to therefore try to figure out an underlying structure in our data. That is, we want to use the data to learn about which observations are more or less similar. Because I do not know what the true relationship is, what we are doing is sometimes called “Unsupervised” learning. In contrast, “supervised” learning is when we are actively bringing information in and “supervising” the characterization being done. We will see an example of this in a few lectures, but for now we are going to start with an unsupervised approach.

Efforts to characterize the relationship within data to determine which observations cluster together (or are segmented) is often an important first step for determining the empirical regularity of interest.

This is what dating sites (e.g., e-harmony) do when they try to figure out which individuals are more or less similar. This is what Facebook and Tik-Tok does when it tries to determine what to show you in your feed. This is what Netflix does when recommending your next series to watch. Personality tests and profiles are another example of this. These tools are also used in marketing to identify likely consumers and by political campaigns to figure out which voters should be targeted and perhaps even how. What they actually do is obviously more complicated, but the basic idea is very, very similar to what we are going to learn today.

# Measurement is Sometimes Discovery

One thing that will become quickly apparent is that how we measure something can have profound implications on what it means - especially if we have no theory to guide us in the organization/analysis of data. Sometimes data exploration = measurement = discovery.

It is also important to note that nothing we are doing is causal – the algorithm is silent as to why the relationships exist. It is equally important to note that the analysis is descriptive, not predictive. This is a critical point with profound implications – if you identify segments in your data and they take proactive steps using that information, the steps you take make affect how future data is clustered.

- Clustering algorithm: discover groups of observations ``similar” to each other.
- Unsupervised learning vs. Supervised learning.
- Descriptive and exploratory data analysis.

The algorithm we are going to use is one of the earliest implementations of clustering and it is very simple in what it does. There are many more complicated procedures and models, but for the purposes of illustrating the general idea (and also the generic limitations of this kind of "unsupervised learning") it is easiest to start with what is perhaps one of the simplest clustering methods.

The procedure used by the k-means clustering algorithm consists of several steps"

1. The data scientist chooses the number of clusters/segments they wish to identify in the data of interest. The number of clusters is given by `K` – hence the name `kmeans`.
2. The computer randomly chooses a initial centroids of `K` clusters in the multidimensional space (where the number of dimensions is the number of variables).
3. Given the choice of `K` centroids, the computer assigns each observation to the cluster whose centroid is the closest (in terms of Euclidian distance).
4. Given that assignment, the computer computes a new centroid for each cluster using the within-cluster mean of the corresponding variable.
5. Repeat Steps 3 and 4 until the cluster assignments no longer change.

So, if there are two variables, say $X$ and $Y$ and we are fitting 2 centroids, the computer will begin by randomly choosing a "centroid" for each cluster – which is simply a point in $(x, y)$. Say $(x_1, y_1)$ for cluster 1 and $(x_2, y_2)$ for cluster 2. Then given this choice, the computer figures out which centroid is "closest" to each data point. So for data point $i$ that is located at $(x_i, y_i)$ the computer computes the distance from each.

Given this, the Euclidean distance to cluster 1 is simply:

$$(x_1 - x_i)^2 + (y_1 - y_i)^2$$

And the Euclidean distance to cluster 2:

$$(x_2 - x_i)^2 + (y_2 - y_i)^2$$

(Note that if we have more variables we just include them in a similar fashion.) Having calculcated the distance to each of the $K$ centroids – here 2 – we now assign each datapoint to either cluster "1" or "2" depending on which is smaller. After doing this for every data point, we then calculate a new centroid by taking the average of all of the points in each cluster in each variable. So if there are $n_1$ observations allocated to cluster 1 and $n_2$ observations allocated to cluster 2 we would compute the new centroids using:

$$x_1 = \sum_i^{n_1} \frac{x_i}{n_1}$$

$$y_1 = \sum_i^{n_1} \frac{y_i}{n_1}$$

$$x_2 = \sum_i^{n_2} \frac{x_i}{n_2}$$

$$y_2 = \sum_i^{n_2} \frac{y_i}{n_2}$$

Now, using these new values for $(x_1, y_1)$ for the centroid of cluster 1 and $(x_2, y_2)$ for the centroid of cluster 2 we reclassify all the observations to allocate each observation to the cluster with the closest centroid. We then recalculate the centroid for each cluster after this reallocation and then we and iterate over these two steps until no

data points change their cluster assignment.

Given this, how sensitive is this to the scale of the variables? What does that imply?

# Applications to Elections and Election Night

Predicting elections requires using votes that are counted to make predictions for ``similar counties.” There are lots of ways to determine similarity based on past voting behavior (and demographics).

Entire books have been written that try to determine how many “Americas” there are. For example…

And many “quizzes” are produced to determine what type of voter you are (more on this later!). For example: quiz from the NY Times (https://www.nytimes.com/interactive/2021/09/08/opinion/republicans-democrats-parties.html).

These are all products that are based on various types of clustering analyses that try to detect pattern in data.

So let’s start simple and think about the task of predicting what is going to happen in a state on Election Night. To do so we want to segment the state into different politically relevant regions so that we can track how well candidates are doing. Or, if you are working for a candidate, which counties should be targeted in get-out-the-vote efforts.

We are going to be working with two datasets. A dataset of votes cast in Florida counties in the 2016 election ( `FloridaCountyData.Rds` ) and also a dataset of the percentage of votes cast for Democratic and Republican presidential candidates in counties (or towns) for the 2004, 2008, 2012, 2016, and 2020 elections ( `CountyVote2004_2020.Rds` ).

To begin, let’s start with Florida in 2016.

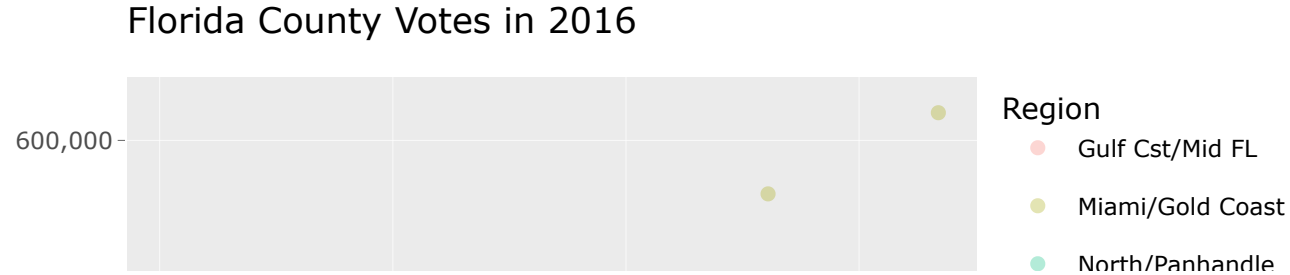
```
library(tidyverse)
library(tidymodels)
library(plotly)

dat <- read_rds(file="https://github.com/jbisbee1/DS1000_S2024/raw/main/data/FloridaCountyData.Rds")
glimpse(dat)
```

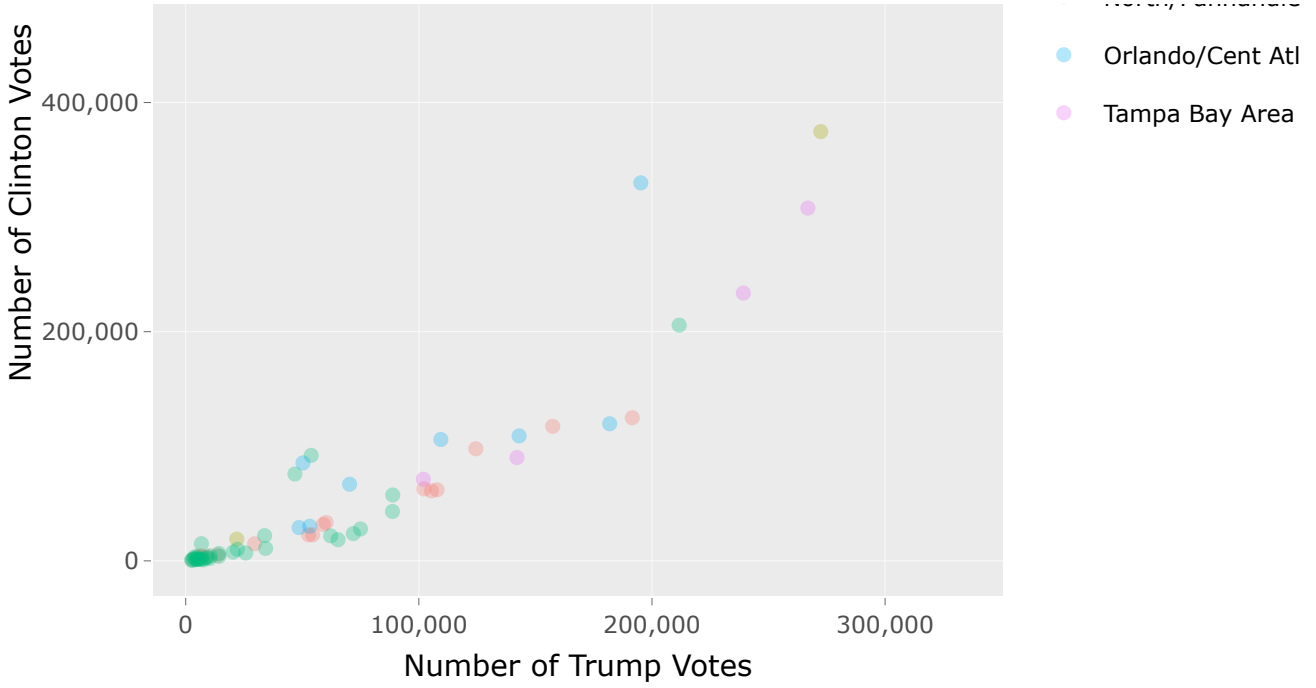
```
## Rows: 67
## Columns: 15
## $ fips_code          <int> 12001, 12003, 12005, 12007, 12009, 12011, 12013, 12…
## $ county_name        <chr> "Alachua", "Baker", "Bay", "Bradford", "Brevard", "…
## $ eligible_voters    <int> 173993, 15092, 118344, 16163, 411191, 1141360, 8620…
## $ party_stratum      <int> 2, 5, 5, 5, 4, 1, 5, 5, 5, 5, 5, 5, 1, 5, 5, 3, 4, …
## $ party_stratum_name <chr> "Mod Democrat", "High Republican", "High Republican…
## $ geo_stratum_name   <chr> "North/Panhandle", "North/Panhandle", "North/Panhan…
## $ Trump              <int> 46834, 10294, 62194, 8913, 181848, 260951, 4655, 60…
## $ Clinton            <int> 75820, 2112, 21797, 2924, 119679, 553320, 1241, 334…
## $ Johnson            <int> 4059, 169, 2652, 177, 9451, 11078, 124, 1946, 1724,…
## $ Stein              <int> 1507, 30, 562, 47, 2708, 5094, 25, 567, 480, 571, 7…
## $ geo_strata         <fct> North/Panhandle, North/Panhandle, North/Panhandle, …
## $ Total2012          <int> 128569, 12634, 87449, 12098, 314744, 831950, 6081, …
## $ Total2016          <int> 128220, 12605, 87205, 12061, 313686, 830443, 6045, …
## $ PctTrump           <dbl> 0.3652628, 0.8166601, 0.7131931, 0.7389934, 0.57971…
## $ PctClinton         <dbl> 0.5913274, 0.1675526, 0.2499513, 0.2424343, 0.38152…
```

The first task that we face as data scientists is to determine which variables are relevant for clustering. Put differently, which variables define the groups we are trying to find. `kmeans` is a very simple algorithm and it assumes that every included variable is equally important for the clustering that is recovered. As a result, if you include only garbage/irrelevant data the relationships you find will also be garbage. The alogorithm is unsupervised in that it has no idea which variables are more or less valuable to what you are trying to find. It is simply trying to find how the data clusters together given the data you have given it! It cannot evaluate the quality of the data you provide.

As a result, when doing `kmeans` we often start with simple visualization around data that we think is likely to be of interest. To make it interactive we can again use `plotly` package and include some `text` information in the `ggplot aes` thetic. We will also clean up the labels and override the default of scientific notation. We will also change the name of the legend to make it descriptive and interpretable.

```
gg<- dat %>%
    ggplot(aes(x = Trump, y = Clinton, color = geo_strata,
               text=paste(county_name))) +
  geom_point(alpha = 0.3) +
  scale_x_continuous(labels=comma) +
  scale_y_continuous(labels=comma) +
  labs(x="Number of Trump Votes",
       y="Number of Clinton Votes",
       title="Florida County Votes in 2016",
       color = "Region")

ggplotly(gg,tooltip = "text")
```

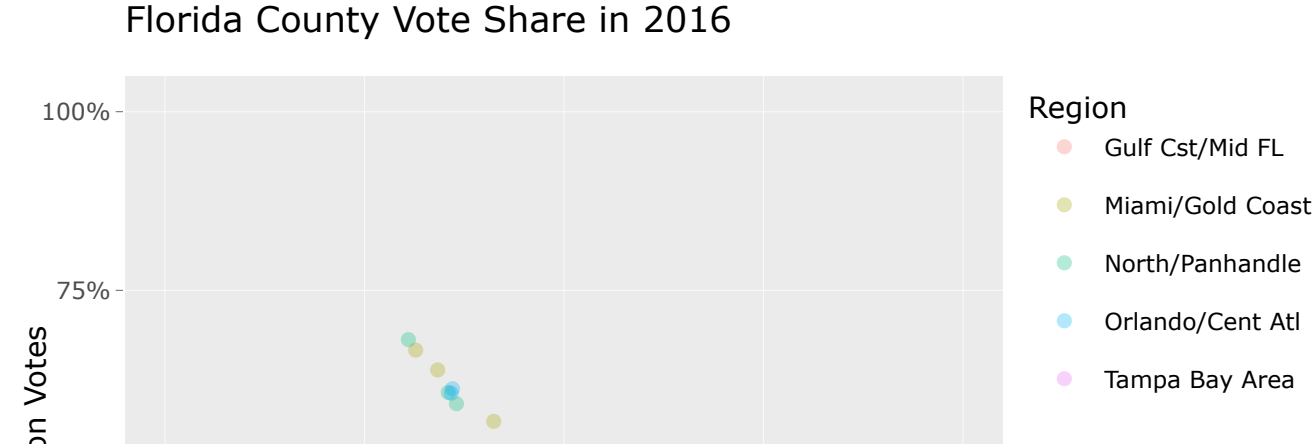


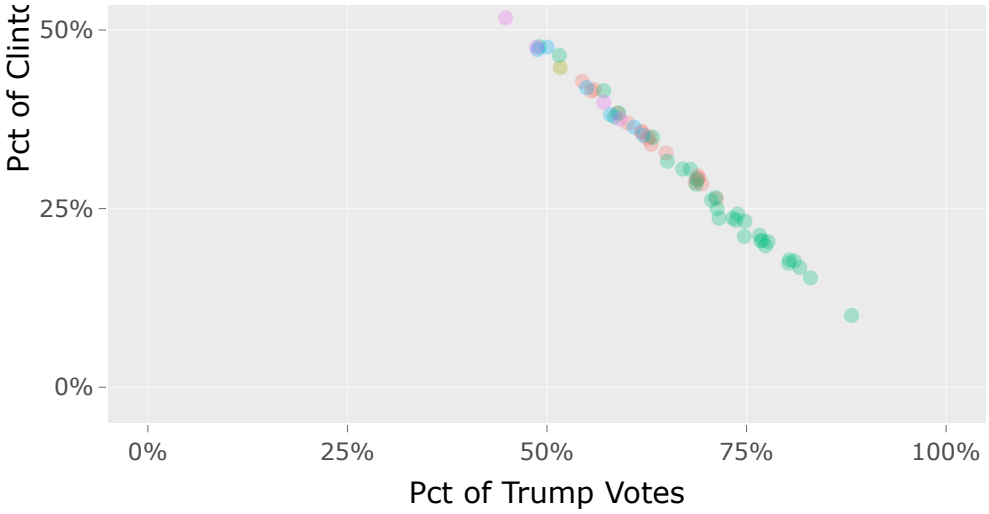



So this suggests that counties with more Clinton votes tend to covary with counties with more Trump voters. Obviously. So we have discovered that there are more votes for both candidates in larger counties. So if we were to cluster based on this we would essentially find groups based on population size. Not useful.

So maybe we should look at the percentage of votes rather than the number of votes. Let's see. Again let's improve the labels and axes and make it interactive using `plotly` to show how the code differs from the default syntax above.

```
gg <- dat %>%
  ggplot(aes(PctTrump, PctClinton, color = geo_strata,
             text=paste(county_name))) +
  geom_point(alpha = 0.3) +
  scale_y_continuous(limits = c(0,1),labels = scales::percent_format(accuracy = 1)) +
  scale_x_continuous(limits = c(0,1),labels = scales::percent_format(accuracy = 1)) +
  labs(x="Pct of Trump Votes",
       y="Pct of Clinton Votes",
       title="Florida County Vote Share in 2016",
       color = "Region")

ggplotly(gg,tooltip = "text")
```

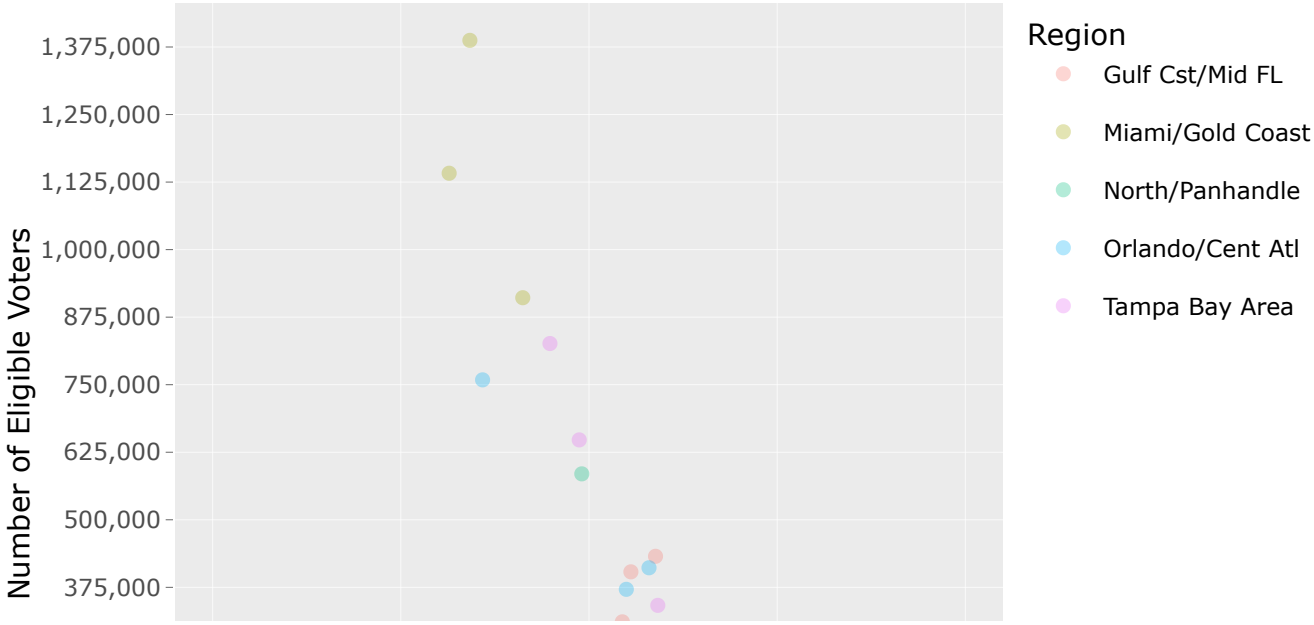


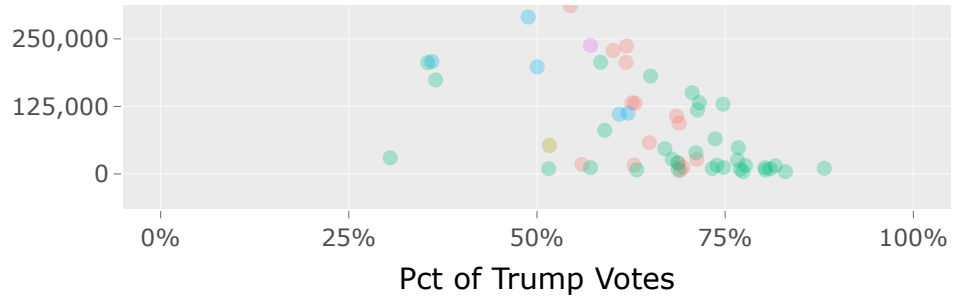



So now we see that places with a higher percentage of support for Clinton have a lower support for Trump. That seems useful if we are interested in characterizing the political context of a county.

But those are highly correlated? Do we need both percentage that support Clinton and also the percentage that support Trump? It seems like that is the same information being "double-counted." What if we include something like the number of eligible voters?

```
gg <- dat %>%
  ggplot(aes(PctTrump, eligible_voters, color = geo_strata,
              text=paste(county_name))) +
  geom_point(alpha = 0.3) +
  scale_y_continuous(label=comma, breaks=seq(0,2000000,by=125000)) +
  scale_x_continuous(limits = c(0,1),labels = scales::percent_format(accuracy = 1)) +
  labs(x="Pct of Trump Votes",
       y="Number of Eligible Voters",
       main="Florida County Election Results in 2016",
       color = "Region")

ggplotly(gg,tooltip = "text")
```






Now things get trickier. Do we want to segment based on the number of eligible voters? Or is it more useful to focus on support for Clinton and Trump? This decision is hugely consequential for the groups `kmeans` will recover. This again highlights the role of the data scientist – *you* need to make a decision and justify it because the decision will be consequential!

To start let's characterize counties by support for Clinton and Trump.

```
rawvote <- dat %>%
  select(c(PctTrump,PctClinton)) %>%
  drop_na()
```

Now we run by providing the data frame of all numeric data and the number of clusters – here `centers` that we want the algorithm to find for us.

```
fl.cluster1 <- kmeans(rawvote, centers=2)
```

Now call the object to see what we have just created and all of the objects that we can now work with.

```
fl.cluster1
```

```
## K-means clustering with 2 clusters of sizes 42, 25
##
## Cluster means:
##    PctTrump PctClinton
## 1 0.7073845  0.2679218
## 2 0.4780951  0.4927738
##
## Clustering vector:
##  [1] 2 1 1 1 2 2 1 1 1 1 1 1 2 1 1 2 2 2 1 2 1 1 1 1 1 2 1 1 2 1 1 1 2 1 1 2 2 1
## [39] 1 2 2 1 1 2 1 1 1 2 2 2 1 2 2 1 1 2 1 2 2 1 1 1 1 2 1 1 1
##
## Within cluster sum of squares by cluster:
## [1] 0.4019281 0.4586045
##  (between_SS / total_SS =  65.3 %)
##
## Available components:
##
## [1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss"
## [6] "betweenss"    "size"         "iter"         "ifault"
```

Alternatively, we can also call the `tidy` function to produce a tibble of the overall fit:

```
clusters <- tidy(fl.cluster1)
clusters
```

```
## # A tibble: 2 × 5
##   PctTrump PctClinton  size withinss cluster
##      <dbl>      <dbl> <int>    <dbl> <fct>
## 1    0.707      0.268    42    0.402 1
## 2    0.478      0.493    25    0.459 2
```

In this object you can see the mean value for each variable in each cluster – i.e., the centroid – as well as the number of observations (here counties) belonging to each cluster, and also the within sum of squares for each cluster. Recall that the centroid for each cluster is simply the average value of the variable for all counties that are assigned to that cluster. (This is the same as the $x_1$, $y_1$, $x_2$, and $y_2$ defined above.)

The values associated with `withinss` are the within sum of squares for all observations in a cluster. This is the sum of the squared distances between each data point in the cluster and the centorid of that cluster. So if $T_i$ denotes the value of `PctTrump` for county $i$ and $C_i$ denotes the value of `PctClinton` for county $i$ the within sum of squares for the $n_1$ counties that belong to cluster $1$ is given by:

$$\sum_{i}^{n_1}(\bar{T}_1 - T_i)^2 + (\bar{C}_1 - C_i)^2$$

if we use $T_1$ to denote the mean of support for Trump in the $n_1$ counties allocated to cluster 1 and $C_1$ to denote the mean support for Clinton in those $n_1$ counties. We will return to this later.

One important thing to note is that `kmeans` starts the algorithm by randomly choosing a centroid for each cluster and then iterating until no classifications change cluster. As a result, the clusters we identify can depend on the initial choices and there is nothing to ensure that the results a "optimal" in a global sense. The classification is conditional on the initial start and the optimization is "local" and relative to that initial choice. So make sure you always set a seed!

*_SELF_TEST* Do a new clustering with 4 centroids called `florida.me` and look at the contests of each cluster using tidy:

```
florida.me <- kmeans(rawvote, centers=4)
tidy(florida.me)
```

```
## # A tibble: 4 × 5
##   PctTrump PctClinton  size withinss cluster
##      <dbl>      <dbl> <int>    <dbl> <fct>
## 1    0.362      0.610     9   0.0359 1
## 2    0.572      0.400    26   0.110  2
## 3    0.792      0.186    13   0.0306 3
## 4    0.697      0.277    19   0.0263 4
```

Because `kmeans` is choosing random start values to start the classification the start values will matter, especially when you are fitting a lot of clusters to a high dimensional dataset (i.e., lots of variables). Even when you are fitting a model with few clusters and few variables the start values may impact the clustering that is found.

To illustrate this lets analyze the same data using the same number of clusters using a different seed value.

```
set.seed(13469) # set new seed value
fl.cluster2 <- kmeans(rawvote,centers=2) # new clustering
```

Now lets compare how the clusters found in `fl.cluster1` compare to the clusters found in the new clustering ( `fl.cluster2` ) using the `table` function.

```
table(fl.cluster1$cluster, fl.cluster2$cluster) # compare clusters
```

```
##
##      1  2
##   1  0 42
##   2 25  0
```

So we can see the classification is exactly flipped. The observations are still largely clustered into the same clustering, but the labels of those clusters is changed. Even though the same information is recovered in both clusterings, the labels of the clusters has changed. This point is essential for replication!

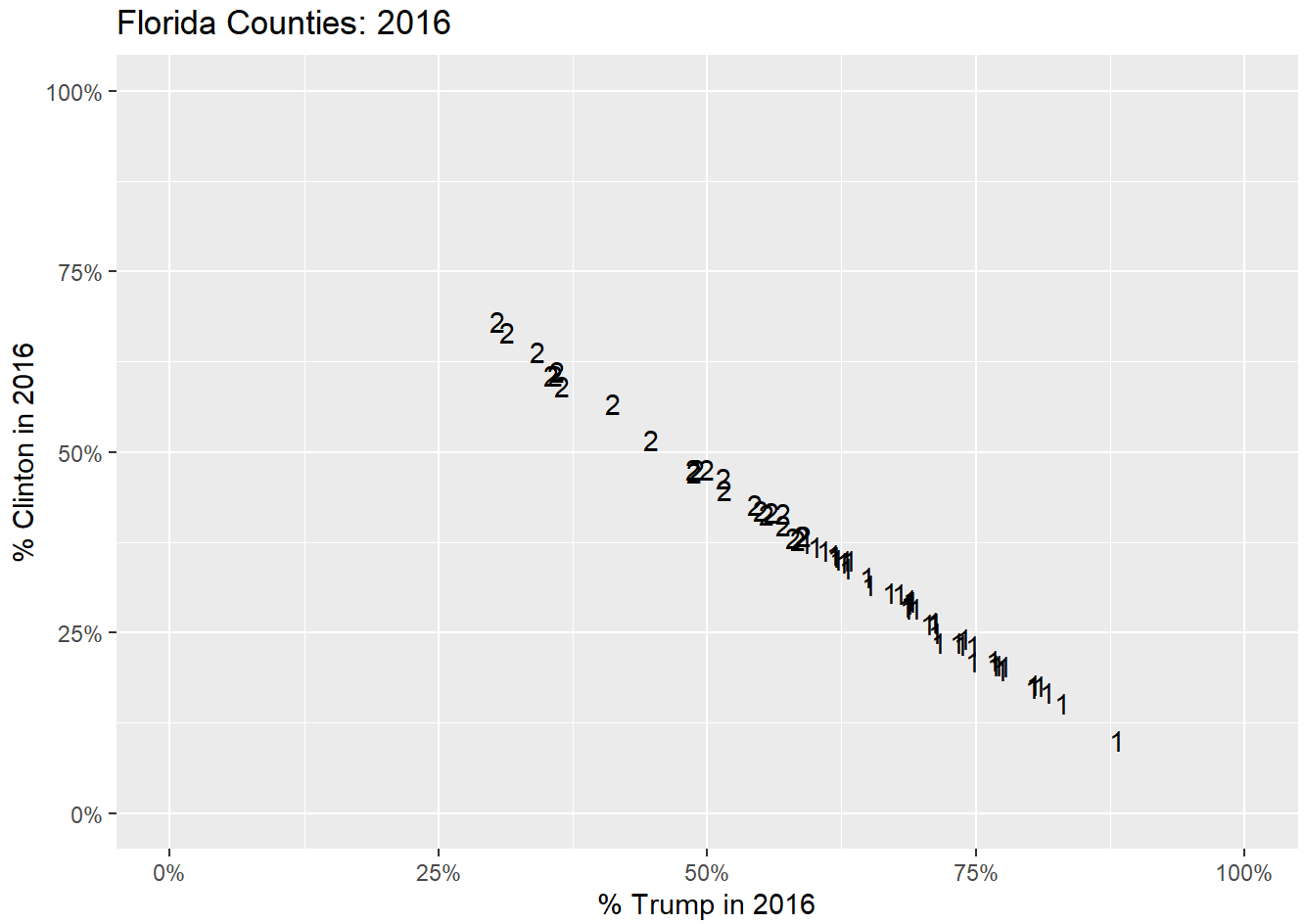
Typically the information we are most interested in is how the observations are clustered – i.e., the labels contained in the `cluster` variable. So how do we get this information back to our original tibble? Thankfully there is a function for that. The `augment` function will add the `cluster` variable from the kmeans clustering onto a tibble containing the data used in the clustering.

```
dat.cluster <- augment(fl.cluster1,dat)
```

With this new augmented tibble – `dat.cluster` – we can now visualize how well the recovered clusters correspond with the underlying data. While this can be more challenging when we are working with high-dimensional data (i.e.., lots of variables) in this case we can visualize the relationship because we have used only two variables in the clustering to characterize the political leanings of counties in Florida.

If I want to plot the points using the cluster label, I can use the `geom_text` code to include the `label` passed to the `ggplot aes` thetic.

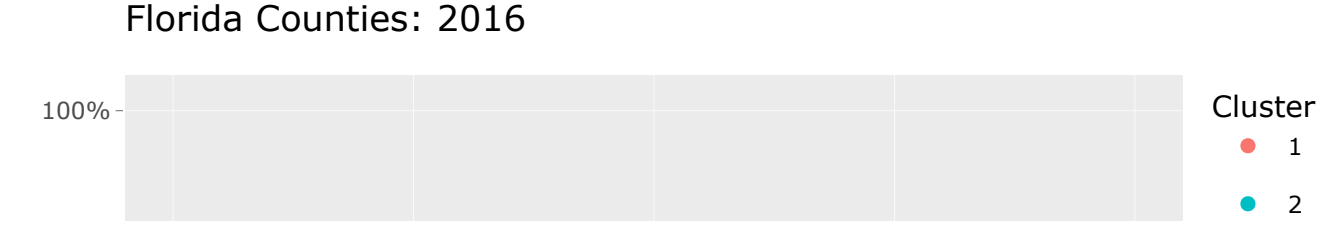
```
dat.cluster %>%
  ggplot(aes(x = PctTrump, y = PctClinton, label = .cluster)) +
  geom_text() +
  scale_y_continuous(limits = c(0,1),labels = scales::percent_format(accuracy = 1)) +
  scale_x_continuous(limits = c(0,1),labels = scales::percent_format(accuracy = 1)) +
  labs(title = "Florida Counties: 2016",
       x = "% Trump in 2016",
       y = "% Clinton in 2016")
```

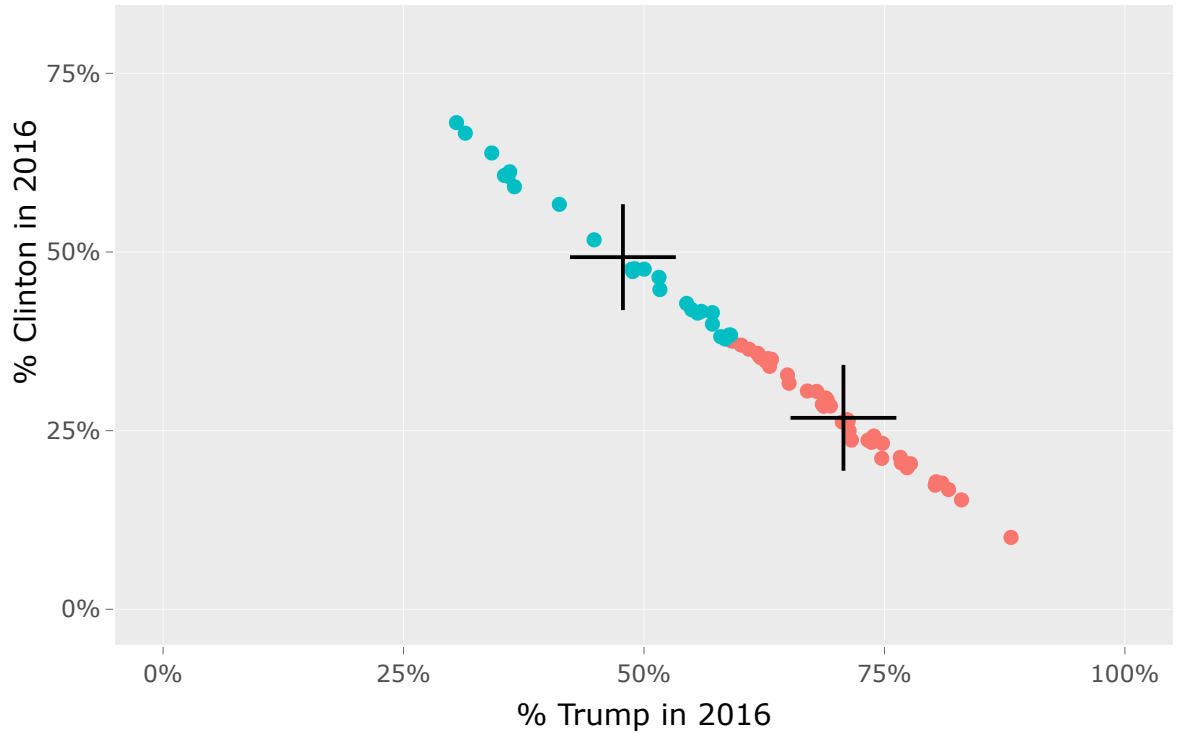



But maybe that is too messy. The overlapping numbers is a bit distracting.

So let's switch to colored points and add in the location of the centroids. This is useful for reminding us of what `kmeans` is actually doing. Let us pull this information from the `clusters` object we created using the `tidy()` function applied to our `kmeans` object. Recall that the location of the centroid is just the average of every variable being analyzed. NOTE: what happens if we replace `color` with `fill` ?

```
gg <- ggplot() +
  geom_point(data=dat.cluster, aes(x = PctTrump, y = PctClinton, color = .cluster,
                                   text=paste(county_name))) +
  geom_point(data=clusters, aes(x = PctTrump, y = PctClinton), size = 10, shape = "+") +
  labs(color = "Cluster",
    title = "Florida Counties: 2016",
       x = "% Trump in 2016",
       y = "% Clinton in 2016") +
  scale_y_continuous(limits = c(0,1),labels = scales::percent_format(accuracy = 1)) +
  scale_x_continuous(limits = c(0,1),labels = scales::percent_format(accuracy = 1))

ggplotly(gg,tooltip = "text")
```






Recall that there is no global minimization being done in this algorithm – all we are doing is starting with a randomly chosen centroid and then doing a (local) minimization given those start value. As a result, you can get different classifications with different start values. Here is a simple example that again shows the sensitivity to start values.

```
set.seed(42)
fl.cluster1 <- kmeans(rawvote,centers=2)

set.seed(13469)
fl.cluster2 <- kmeans(rawvote,centers=2)

table(fl.cluster1$cluster,fl.cluster2$cluster)
```

```
##
##      1  2
##   1 21  0
##   2  4 42
```

But we can use `nstart` to try multiple initial configurations and use the one that produces the best total within sum of squares given the number of centers being chosen. Given that we are only classifying based on two variables let's try 25 different start values.

```
set.seed(42)
fl.cluster1 <- kmeans(rawvote,centers=2,nstart=25)

set.seed(13469)
fl.cluster2 <- kmeans(rawvote,centers=2,nstart=25)

table(fl.cluster1$cluster,fl.cluster2$cluster)
```

```
##
##      1  2
##   1  0 21
##   2 46  0
```

So now you can see that using multiple start values eliminates the classification differences based on the initial start value! Now the clusters have the same counties in each – although with different names. What `nstart` is doing is having the algorithm try a bunch of different start values and then choose the centroid that has the lowest within sum of squares as the starting value. So while this is not doing a search over every possible start value, it chooses the "best" start value among the set of values it generates.

Now think about doing a `kmeans` for three clusters. Based on the figure we just created, where do you think the three clusters will be located.

***SELF TEST*** Now implement this! What do you observe? Were you correct?

```
# INSERT CODE HERE
```

# The variables you use matter!

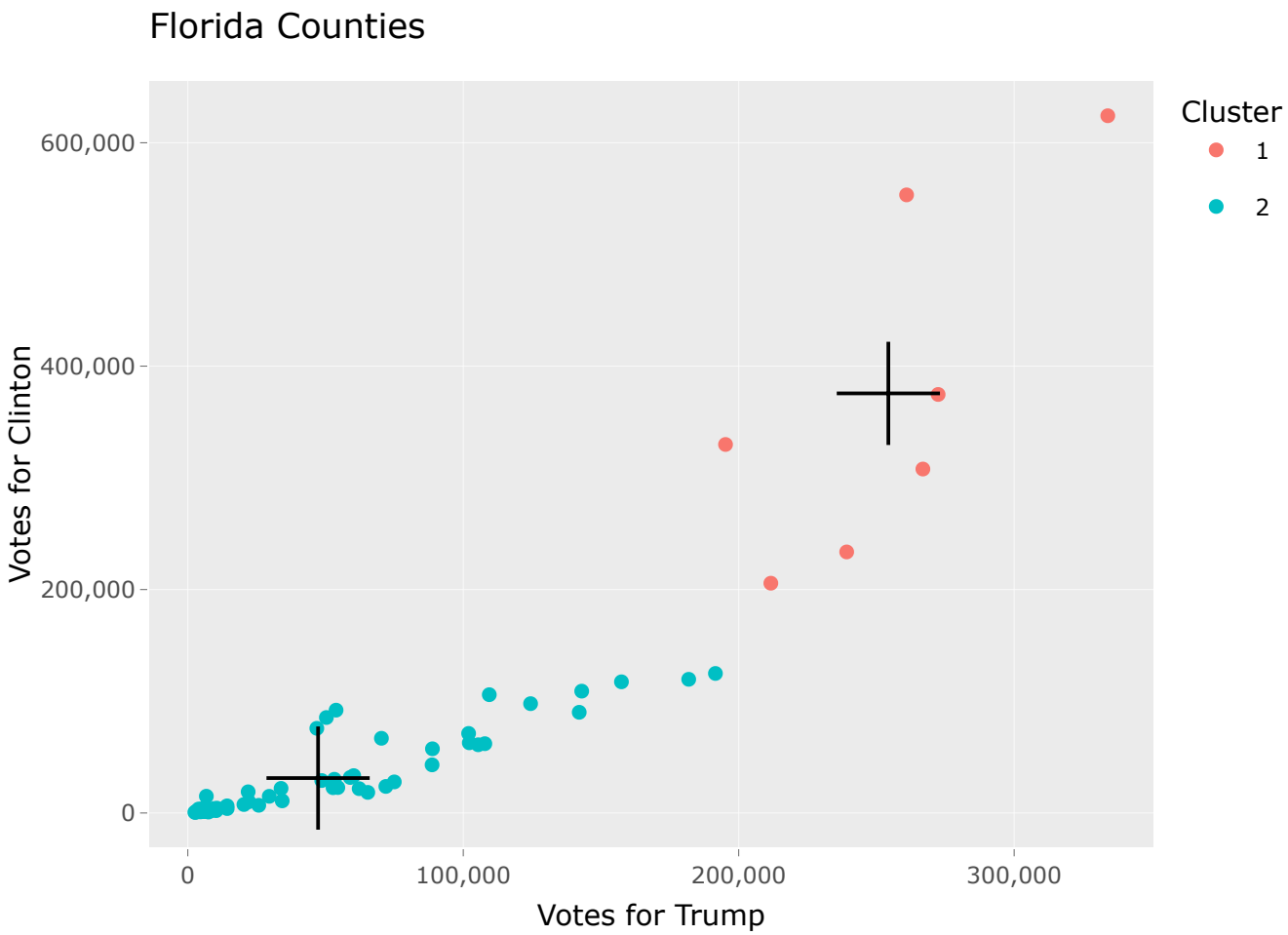
So what if we did cluster based on the number of votes cast? How would that affect the conclusions we get? Instead of clustering based on `PctTrump` and `PctClinton` do the clustering using `Trump` and `Clinton` . Can you predict what will happen before you do it?

```
rawvote <- dat %>%
  select(c(Trump,Clinton)) %>%
  drop_na()

fl.cluster1.count <- kmeans(rawvote, centers=2)

dat.cluster2 <- augment(fl.cluster1.count,dat)
clusters <- tidy(fl.cluster1.count)

clusters
```

```
## # A tibble: 2 × 5
##     Trump Clinton  size      withinss cluster
##     <dbl>   <dbl> <int>         <dbl> <fct>
## 1 254330. 375619.     7 161451108936. 1
## 2  47293.  31261.    60 226694181010. 2
```

Now graph the new clusters, labeling the county names.

```
gg <- ggplot() +
  geom_point(data=dat.cluster2, aes(x = Trump, y = Clinton, color = .cluster,
                                    text=paste(county_name))) +
  geom_point(data=clusters, aes(x = Trump, y= Clinton), size = 10, shape = "+") +
  labs(color = "Cluster",
       title = "Florida Counties",
       x = "Votes for Trump",
       y = "Votes for Clinton") +
  scale_y_continuous(label=comma) +
  scale_x_continuous(label=comma)

ggplotly(gg,tooltip = "text")
```




And finally, how do the clusters compare to one another? If you do a table of the clusters against one another what do you observe?

```
table(ByPct = fl.cluster1$cluster,
      ByVote= fl.cluster1.count$cluster)
```

```
##      ByVote
## ByPct  1  2
##     1  7 14
##     2  0 46
```

The scale matters. What if we do a clustering using `eligible voters` , `PctTrump` and `PctClinton` . What do you observe for a clustering of these variables using `k=2` clusters?

```
rawvote3 <- dat %>%
  select(c(eligible_voters,PctClinton,PctTrump))

cluster.mix <- kmeans(rawvote3,centers=2,nstart=25)
tidy(cluster.mix)
```

```
## # A tibble: 2 × 6
##   eligible_voters PctClinton PctTrump  size      withinss cluster
##             <dbl>      <dbl>    <dbl> <int>         <dbl> <fct>
## 1         110305.      0.327    0.647    60 842735627916. 1
## 2         893950       0.564    0.407     7 483680203618. 2
```

Now rescale the data being fit using the `scale` function to normalize the data to have mean 0 and variance 1. (Note that `scale` just normalizes a data.frame object.) Why is this important given the alogorithm being used? Now cluster the rescaled data. How do the resulting clusters compare to the unrescaled clusters and also our original scaling based on the percentages – `fl.cluster1` ?

```
rawvote3.scale <- scale(rawvote3)
summary(rawvote3.scale)
```

```
##  eligible_voters      PctClinton          PctTrump
##  Min.   :-0.67089   Min.   :-1.84762   Min.   :-2.29670
##  1st Qu.:-0.62953   1st Qu.:-0.77653   1st Qu.:-0.50178
##  Median :-0.35021   Median :-0.02995   Median : 0.06301
##  Mean   : 0.00000   Mean   : 0.00000   Mean   : 0.00000
##  3rd Qu.: 0.09304   3rd Qu.: 0.48713   3rd Qu.: 0.67141
##  Max.   : 4.26750   Max.   : 2.42012   Max.   : 1.88340
```

```
cluster.mix2 <- kmeans(rawvote3.scale,centers=2,nstart=25)
tidy(cluster.mix2)
```

```
## # A tibble: 2 × 6
##   eligible_voters PctClinton PctTrump  size withinss cluster
##             <dbl>      <dbl>    <dbl> <int>    <dbl> <fct>
## 1           0.984      1.21    -1.22     20     52.7 1
## 2          -0.419     -0.515    0.518    47     33.7 2
```

How compare? Start with the normalized vs. unnormalized clustering (i.e., same variables but different scale).

```
table(Normalized = cluster.mix2$cluster,
      Unnormalized = cluster.mix$cluster)
```

```
##           Unnormalized
## Normalized  1  2
##          1 13  7
##          2 47  0
```

Now compare to the original clustering we did using vote share (i.e., different variables and different scale).

```
table(Normalized = cluster.mix2$cluster,
      ByPct = fl.cluster1$cluster)
```

```
##           ByPct
## Normalized  1  2
##          1 18  2
##          2  3 44
```

This means…

# More Data! More Clusters ?

Perhaps we need more data. Lets get all of the county (or town) level data from 2004 up through 2020. Let's focus on Florida again.

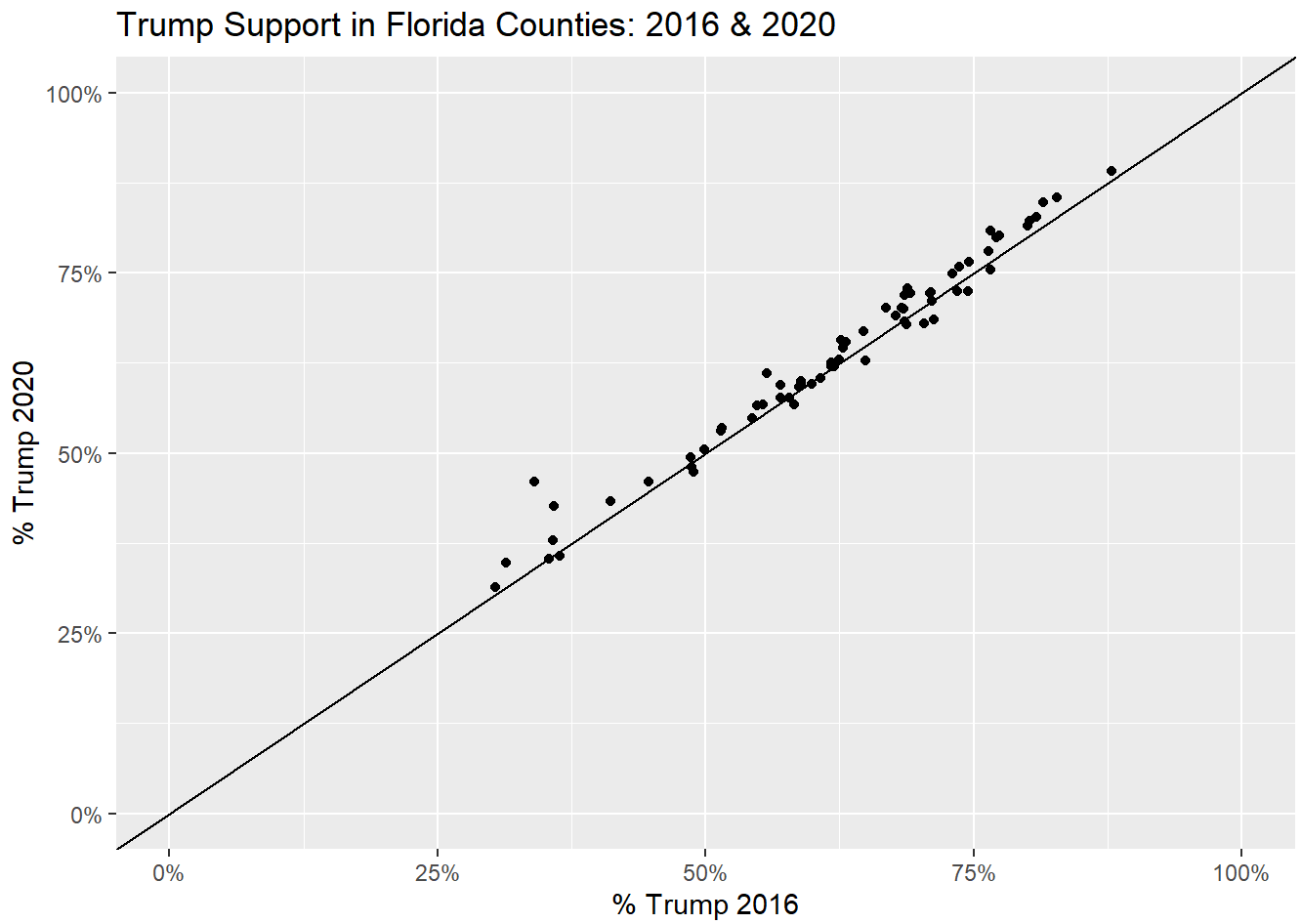
```
dat.all <- read_rds(file="https://github.com/jbisbee1/DS1000_S2024/raw/main/data/CountyVote2004_
2020.Rds")

dat.fl <- dat.all %>%
  filter(state=="FL")
```

For now, let us work with `pct_rep_2016` and `pct_rep_2020` – but try replicating the results using a different choice to see what happens. Note that `kmeans` takes a data frame with all numeric columns so let's start by creating a new tibble with just numeric data and no missingness.

```
rawvote <- dat.fl %>%
  select(c(pct_rep_2004,pct_rep_2008,pct_rep_2012,pct_rep_2016,pct_rep_2020)) %>%
  drop_na()
```

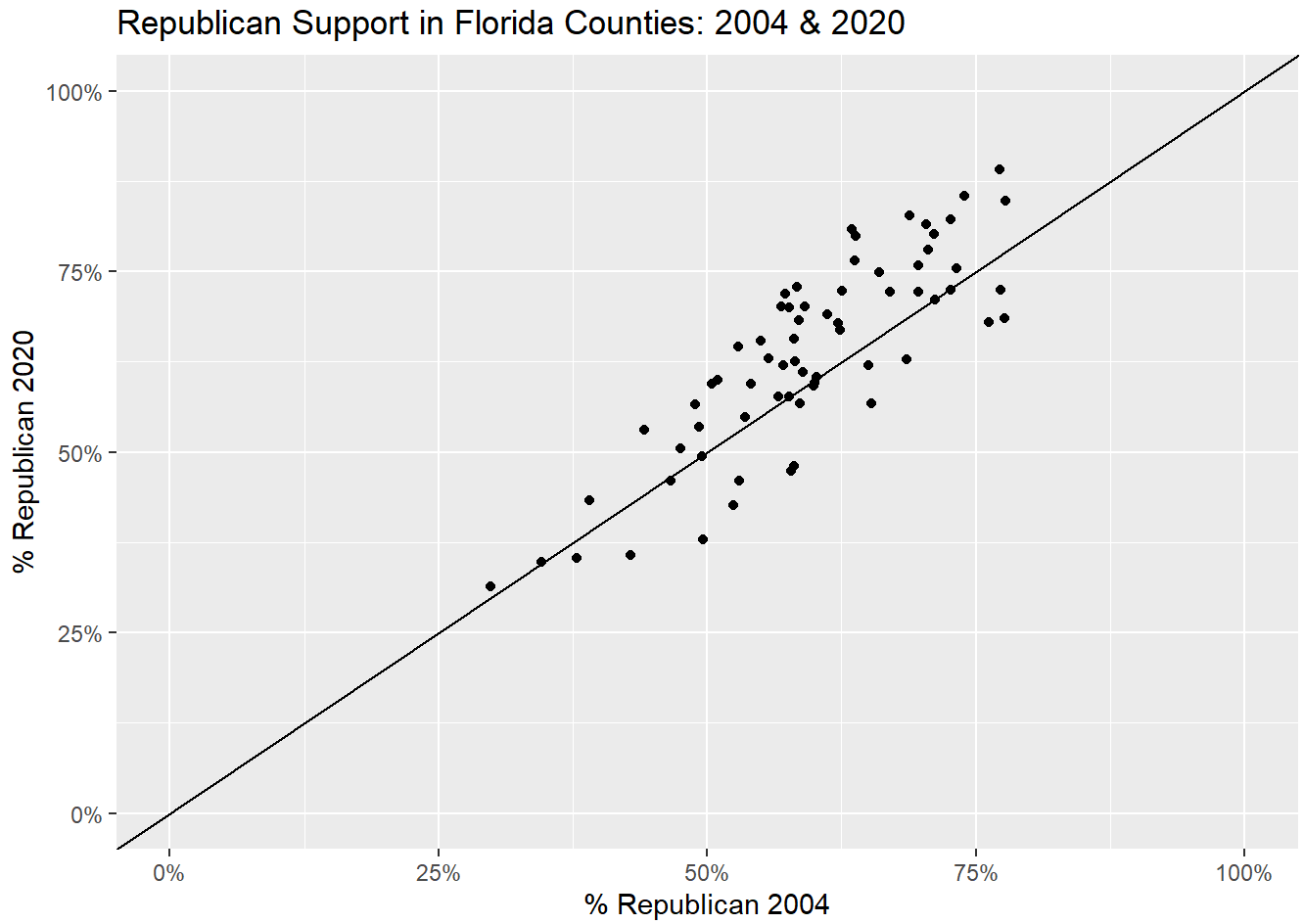
Again we can start by visualizing the relationship. Since we can only think in 2 dimensions, lets' look at some.

```
rawvote %>%
  ggplot(aes(x=pct_rep_2016, y=pct_rep_2020)) +
  geom_point() +
  labs(x="% Trump 2016",
       y = "% Trump 2020",
       title = "Trump Support in Florida Counties: 2016 & 2020") +
  geom_abline(intercept=0,slope=1) +
  scale_x_continuous(limits = c(0,1),labels = scales::percent_format(accuracy = 1)) +
  scale_y_continuous(limits = c(0,1),labels = scales::percent_format(accuracy = 1))
```




What if we compare Republican vote share in 2004 and 2020. What does that show?

```
rawvote %>%
  ggplot(aes(x=pct_rep_2004, y=pct_rep_2020)) +
  geom_point() +
  labs(x="% Republican 2004",
       y = "% Republican 2020",
       title = "Republican Support in Florida Counties: 2004 & 2020") +
  geom_abline(intercept=0,slope=1) +
  scale_x_continuous(limits = c(0,1),labels = scales::percent_format(accuracy = 1)) +
  scale_y_continuous(limits = c(0,1),labels = scales::percent_format(accuracy = 1))
```




# How many clusters?

So a critical question is always – how many clusters should I use? An issue with answering this question is that there really isn't a statistical theory to guide this determination. More clusters will always "explain" more variation, and if we choose the number of clusters equal to the number of data points we will perfectly "fit/explain" the data. But it will be a trivial explanation and not give us any real information. Recall that one of the goals is to use the clustering to reduce the dimensionality of the data in a way that recovers a "meaningful" representation of the underlying data.

So let us explore how the clustering changes for different numbers of centers. What we are going to do is to create a tibble called `kcluster.fl` that is going contain the results of a `kmeans` clustering for 10 different choices of `K` that varies from 1 to 10.

```
kcluster.fl <-
  tibble(K = 1:10) %>%   # define a sequence that we will use to denote k
  mutate(   # now we are going to create new variables in this tibble
    kcluster = map(K, ~kmeans(rawvote, .x, nstart = 25)),   # run a kmeans clustering using k
    tidysummary = map(kcluster, tidy), # run the tidy() function on the kcluster object
    augmented = map(kcluster, augment, rawvote), # save the cluster to the data
  )
```

The above code uses the `map` function which is how we can apply a function across an index. So in line 337 we are going to take `map` to apply the sequence of `K` values we defined running from 1 to 10 the `kmeans()` algorithm applied to the `rawvotes` tibble. The `.x` in the line reveals where we are going to substitute the value being

mapped. So the object `kcluster` is going to be a list of 10 elements – each list element being a `kmeans` object associated with the choice of `K` centers.

We then map the `tidy` function to the list of `kcluster` we just created – creating a list called `tidysummary` where each element is the summary associated with the kth clustering. We next create a tibble that augments the original data being clustered – `rawvotes` with the cluster label associated with the kth clustering. (But remember that the meaning of those labels is not fixed!)

So let's give in to see what we have just done. If we take a look at the first row of `kcluster.fl` we can see that it consists of a vector of the sequence of k's we defined, and then the three list objects we created – `kcluster`, `tidysummary`, and `augmented`.

```
kcluster.fl[1,]
```

```
## # A tibble: 1 × 4
##       K kcluster tidysummary      augmented
##   <int> <list>   <list>           <list>
## 1     1 <kmeans> <tibble [1 × 8]> <tibble [67 × 6]>
```

But to work with these objects we need to extract these lists. Lists are a pain in R – especially when you are starting out – so do not think too hard about the following. What we are going to essentially do is to extract each of the list objects in `kcluster.fl` to a separate tibble.

```
clusters <- kcluster.fl %>%
  unnest(cols=c(tidysummary))

clusters
```

```
## # A tibble: 55 × 11
##        K kcluster pct_rep_2004 pct_rep_2008 pct_rep_2012 pct_rep_2016
##    <int> <list>          <dbl>        <dbl>        <dbl>        <dbl>
##  1     1 <kmeans>        0.595        0.581        0.595        0.620
##  2     2 <kmeans>        0.674        0.680        0.695        0.730
##  3     2 <kmeans>        0.519        0.484        0.499        0.514
##  4     3 <kmeans>        0.580        0.562        0.584        0.620
##  5     3 <kmeans>        0.462        0.420        0.425        0.416
##  6     3 <kmeans>        0.707        0.716        0.727        0.759
##  7     4 <kmeans>        0.601        0.588        0.611        0.648
##  8     4 <kmeans>        0.531        0.495        0.511        0.533
##  9     4 <kmeans>        0.712        0.724        0.734        0.765
## 10     4 <kmeans>        0.416        0.374        0.371        0.351
## # ℹ 45 more rows
## # ℹ 5 more variables: pct_rep_2020 <dbl>, size <int>, withinss <dbl>,
## #   cluster <fct>, augmented <list>
```

So clusters is a tibble that consists of the centroids associated with each of the centroids in each of the `K` clusterings we did.

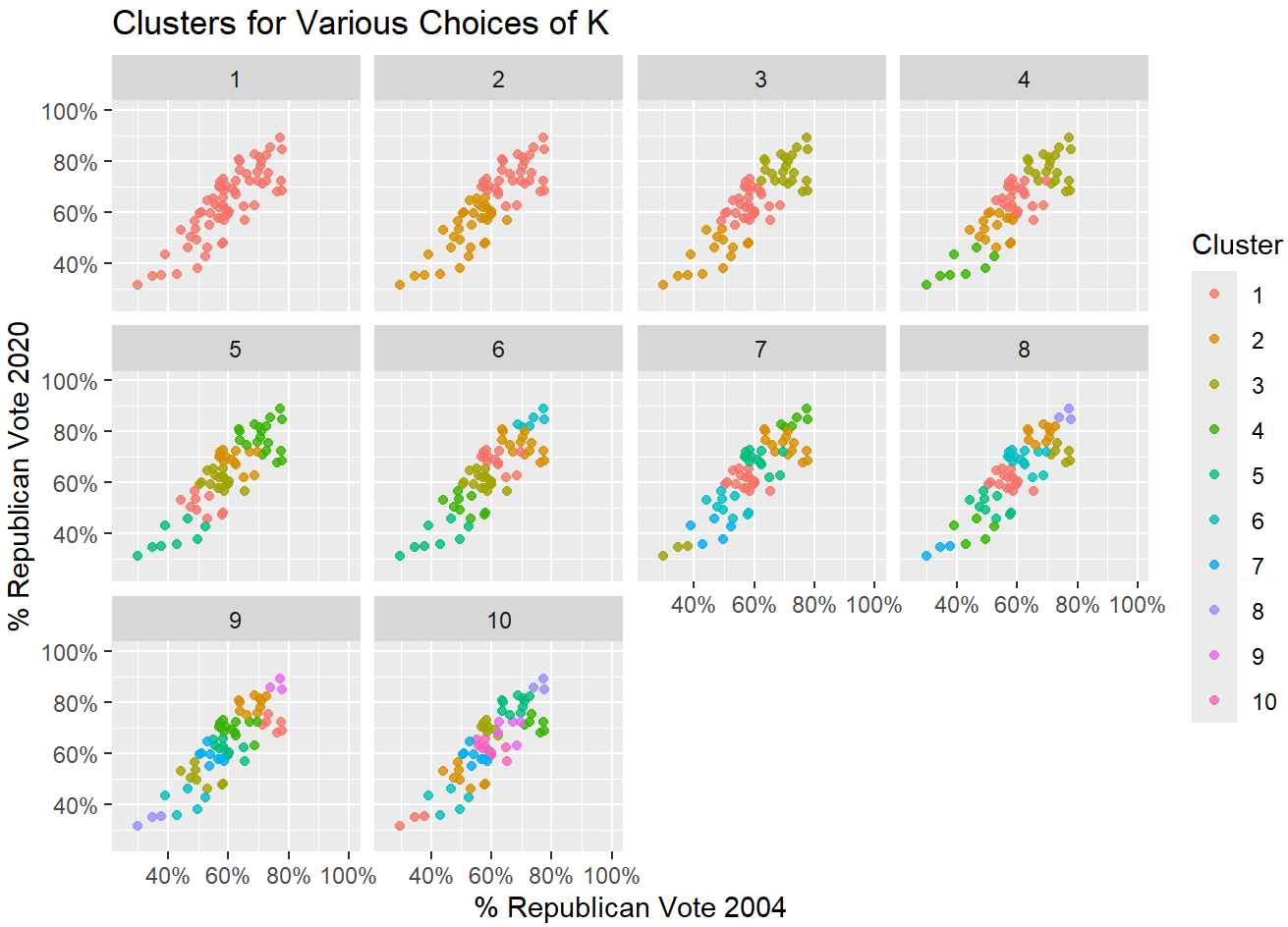
We can also extract the clusters associated with each of the observations by doing a similar operation on the `augmented` list we created.

```
points <- kcluster.fl %>%
  unnest(cols=c(augmented))

points
```

```
## # A tibble: 670 × 9
##        K kcluster tidysummary      pct_rep_2004 pct_rep_2008 pct_rep_2012
##    <int> <list>   <list>                  <dbl>        <dbl>        <dbl>
##  1     1 <kmeans> <tibble [1 × 8]>        0.429        0.386        0.405
##  2     1 <kmeans> <tibble [1 × 8]>        0.777        0.784        0.789
##  3     1 <kmeans> <tibble [1 × 8]>        0.712        0.699        0.712
##  4     1 <kmeans> <tibble [1 × 8]>        0.696        0.697        0.706
##  5     1 <kmeans> <tibble [1 × 8]>        0.577        0.547        0.558
##  6     1 <kmeans> <tibble [1 × 8]>        0.346        0.324        0.323
##  7     1 <kmeans> <tibble [1 × 8]>        0.634        0.696        0.710
##  8     1 <kmeans> <tibble [1 × 8]>        0.557        0.531        0.567
##  9     1 <kmeans> <tibble [1 × 8]>        0.569        0.574        0.604
## 10     1 <kmeans> <tibble [1 × 8]>        0.762        0.711        0.725
## # ℹ 660 more rows
## # ℹ 3 more variables: pct_rep_2016 <dbl>, pct_rep_2020 <dbl>, .cluster <fct>
```

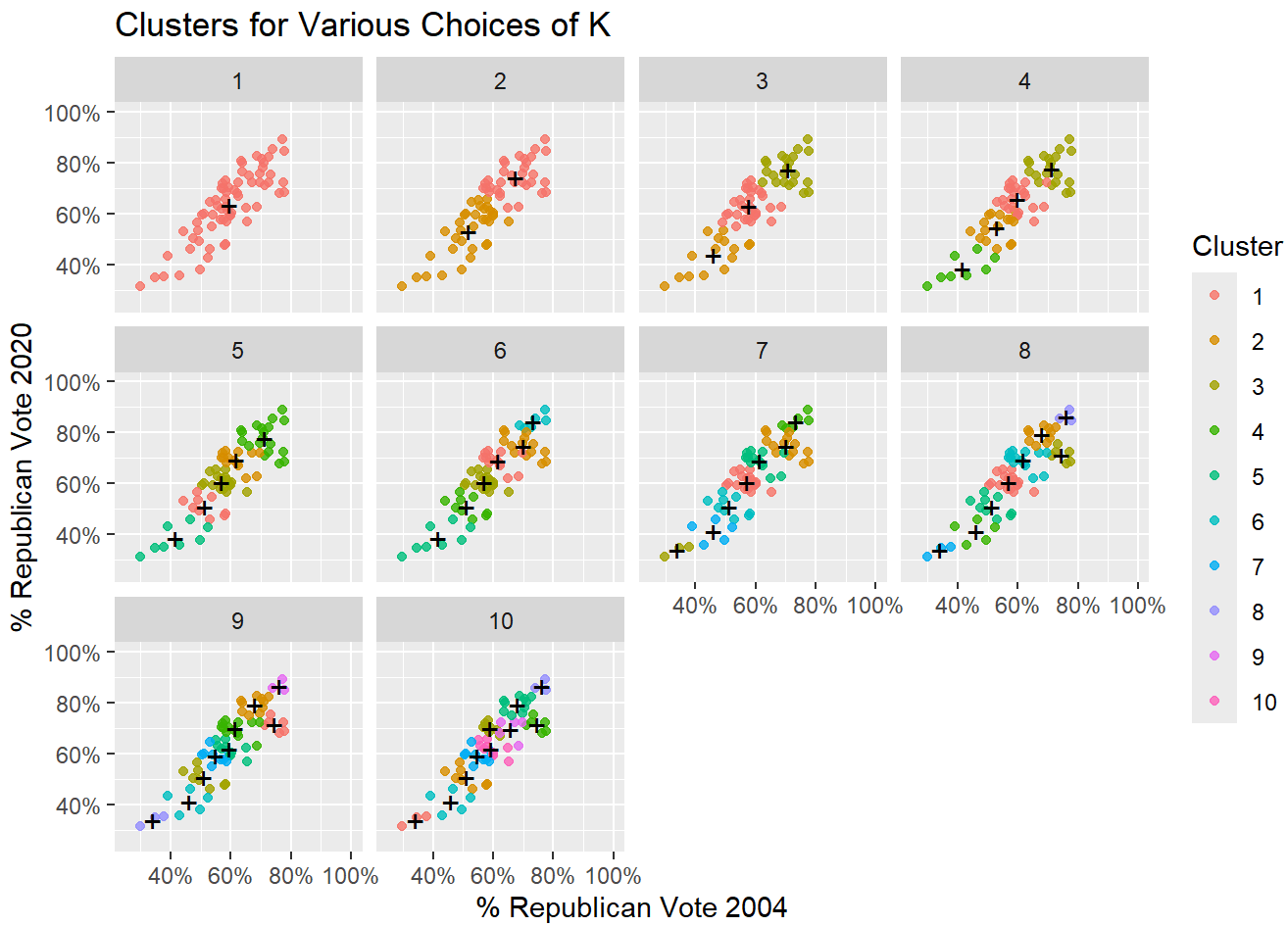
So now we can plot the results. To do so we are going to produce multiple plots by “facet-wrapping” using values `K`.

```
p1 <-
  ggplot(points, aes(x = pct_rep_2004, y = pct_rep_2020)) +
  geom_point(aes(color = .cluster), alpha = 0.8) +
  labs(x = "% Republican Vote 2004",
       y = "% Republican Vote 2020",
       color = "Cluster",
       title = "Clusters for Various Choices of K") +
  facet_wrap(~ K) +
  scale_x_continuous(limits = c(.25,1),labels = scales::percent_format(accuracy = 1)) +
  scale_y_continuous(limits = c(.25,1),labels = scales::percent_format(accuracy = 1))

p1
```




Now let's add in the centroids!

```
p1 + geom_point(data = clusters, size = 4, shape = "+")
```




How does Total Within Sum of squares change as clusters increase? Recall that the within sum of squares for each cluster is simply how far each data point in a cluster is from the centroid according to squared Euclidean distance. Thus, if $T$ denotes `PctTrump` and $C$ denotes `PctClinton` the within sum of squares for cluster $k$ using the $n$ counties that are allocated in cluster $k$ is given by:

$$WSS_k = \sum_i^n (\bar{T}_k - T_i)^2 + (\bar{C}_k - C_i)^2$$

Given this, the total within sum of squares is simply the sum of the within sum of squares across the k clusters. In other words, if we fit $K$ clusters, the total within sum of squares is:

$$TSS = \sum_k^K WSS_k$$

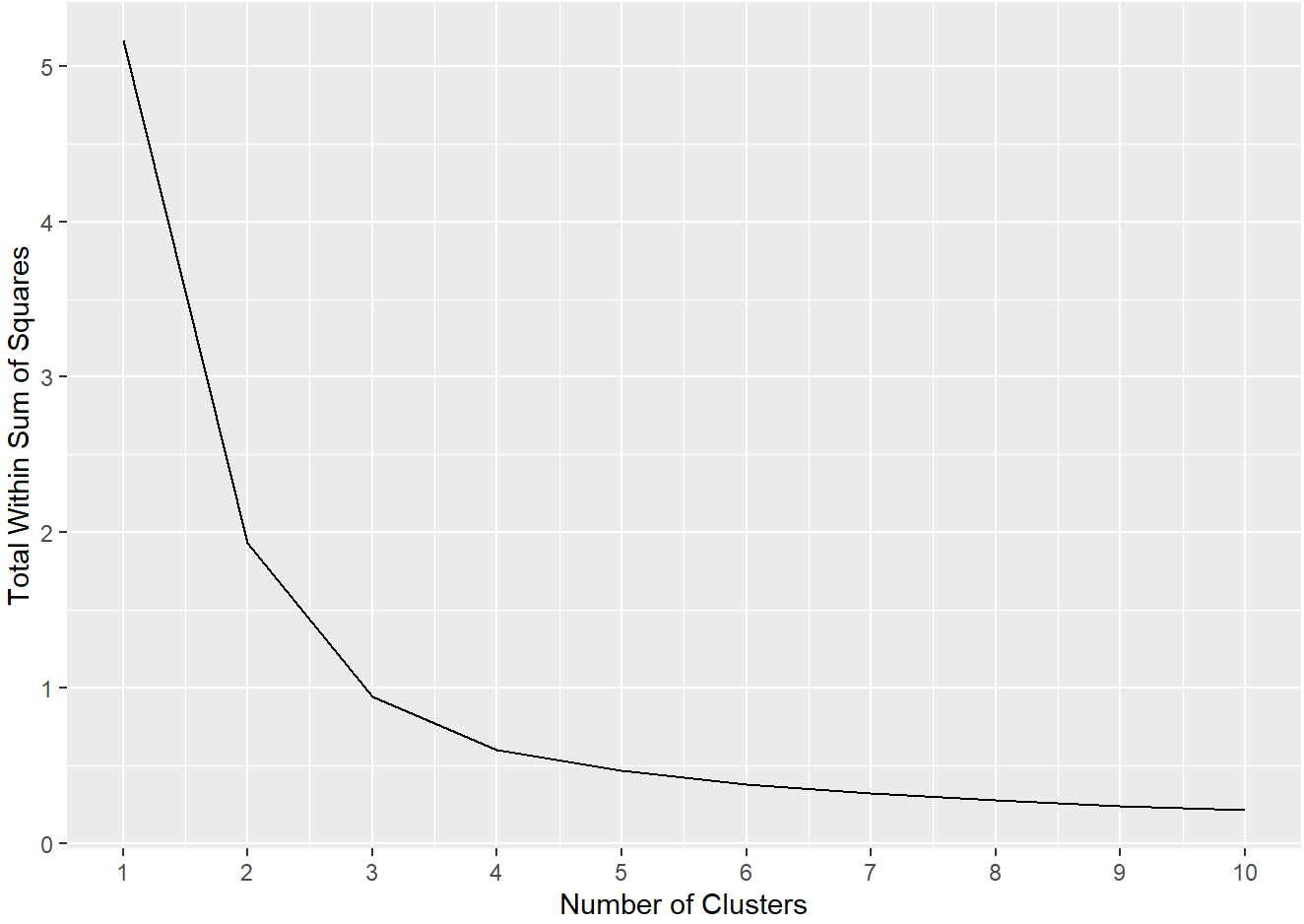
Note that the total sum of squares will usually decrease as the number of clusters increase, Why? Because more clusters means more centroids which will mean smaller squared distances. If, for example, we fit a model with as many centroids as there are observations – i.e., $K == N$ – then the within sum of squares for every observation would be $0$ and the total sum of squares would also be $0$! Note that the total sum of squares will not always decrease depending on number of clusters because of the dependence on start values. Especially when analyzing many variables, the results become more sensitive it is to start values!

Too see what we have created, let us take a look within the tibble `kcluster.fl` and extract the second list item – which is the set of tibbles summarizing the overall fit.

```
fits <- kcluster.fl[[2]]
```

To extract the total within sum-of-squares from this we can write a loop to extract the information. Note that we are using `[[]]` to select an element from a list. Then we can plot the relationship. For simplicity I have used the standard plot command in base R rather than

```
tot.withinss <- NULL

for(i in 1:10){
  tot.withinss[i] <- fits[[i]]$tot.withinss
}

fit <- bind_cols(k = seq(1,10), tot.withinss = tot.withinss)

ggplot(fit, aes(x=k,y=tot.withinss)) +
  geom_line() +
  scale_x_continuous(breaks=seq(1,10)) +
  labs(x="Number of Clusters", y ="Total Within Sum of Squares")
```




# Identifying the meaning of clusters

OK, so how do we interpret what this means? Or label the clusters sensibly? This again requires the data scientist to examine and mutate the data.

```
set.seed(13469)
fl.cluster <- kmeans(rawvote, centers=5, nstart = 25)
dat.cluster <- augment(fl.cluster,dat.fl)
```

```
tidy(fl.cluster) %>%
  arrange(-pct_rep_2020)
```

```
## # A tibble: 5 × 8
##   pct_rep_2004 pct_rep_2008 pct_rep_2012 pct_rep_2016 pct_rep_2020  size
##          <dbl>        <dbl>        <dbl>        <dbl>        <dbl> <int>
## 1        0.714        0.727        0.737        0.768        0.779    19
## 2        0.619        0.617        0.638        0.678        0.692    14
## 3        0.570        0.541        0.562        0.596        0.607    17
## 4        0.513        0.475        0.493        0.503        0.510     9
## 5        0.416        0.374        0.371        0.351        0.384     8
## # ℹ 2 more variables: withinss <dbl>, cluster <fct>
```

Now change the order of the factor so that it is ordered from most Trump supporting to most Clinton Supporting. To do so we need to use the `factor` function to re-define the order of the levels.

```
dat.cluster <- dat.cluster %>%
  mutate(cluster = factor(.cluster,
                          levels=c(3,5,2,4,1)))
```

Now let's check that we did this correctly. Let's see if the clusters are arranged by average Trump support.

```
dat.cluster %>%
  group_by(cluster) %>%
  summarize(PctTrump = mean(pct_rep_2020))
```

```
## # A tibble: 5 × 2
##   cluster PctTrump
##   <fct>      <dbl>
## 1 3          0.779
## 2 5          0.692
## 3 2          0.607
## 4 4          0.510
## 5 1          0.384
```

Yes, but the labels are weird and unintuitive. Let's fix this by using the `factor` function to change the `labels` associated with each factor value.

```
dat.cluster <- dat.cluster %>%
  mutate(cluster = factor(cluster,
                          labels=c("Very Strong Rep","Strong Rep","Rep","Toss Up","Strong De
m")))
```

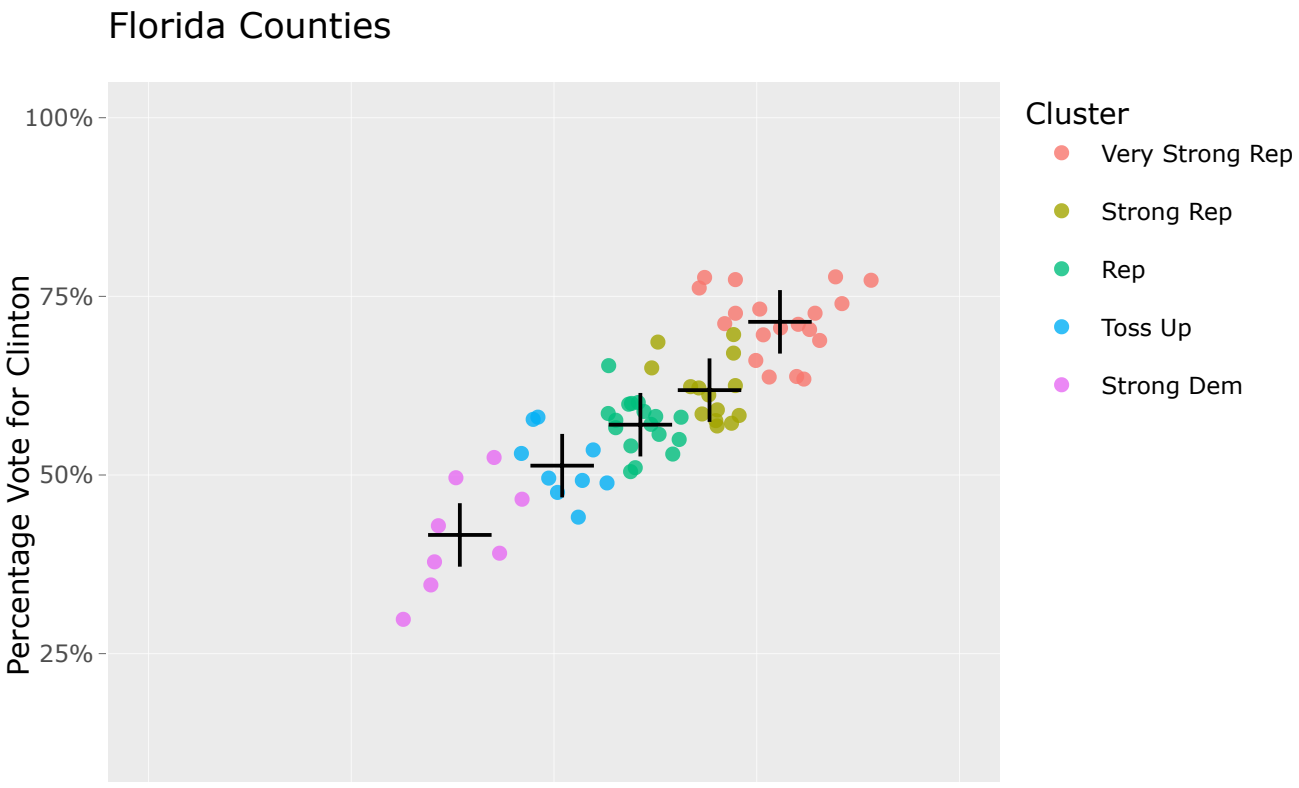
Now confirm that we did not screw that up.

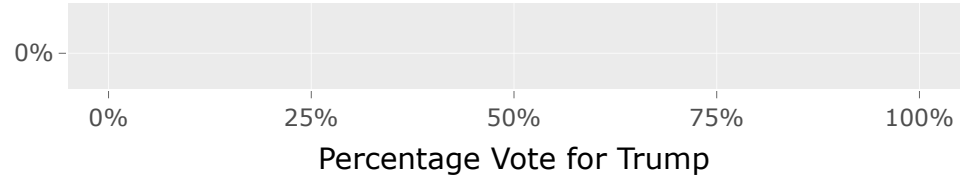
```
dat.cluster %>%
  group_by(cluster) %>%
  summarize(PctTrump = mean(pct_rep_2020))
```

```
## # A tibble: 5 × 2
##   cluster         PctTrump
##   <fct>              <dbl>
## 1 Very Strong Rep    0.779
## 2 Strong Rep         0.692
## 3 Rep                0.607
## 4 Toss Up            0.510
## 5 Strong Dem         0.384
```

This seems good to go.

```
fl.centers <- as.data.frame(fl.cluster$centers)

gg <- ggplot() +
  geom_point(data=dat.cluster, aes(x = pct_rep_2020, y = pct_rep_2004, color = cluster,
                                   text=paste(county.name)), alpha = 0.8) +
  geom_point(data=fl.centers, aes(x = pct_rep_2020, y = pct_rep_2004), size = 6, shape = "+") +
  labs(color = "Cluster",
       title = "Florida Counties",
       x = "Percentage Vote for Trump",
       y = "Percentage Vote for Clinton") +
  scale_y_continuous(limits = c(0,1),labels = scales::percent_format(accuracy = 1)) +
  scale_x_continuous(limits = c(0,1),labels = scales::percent_format(accuracy = 1))

ggplotly(gg,tooltip = "text")
```






So how well does our classification compare to the one that was used by the networks on Election Night?

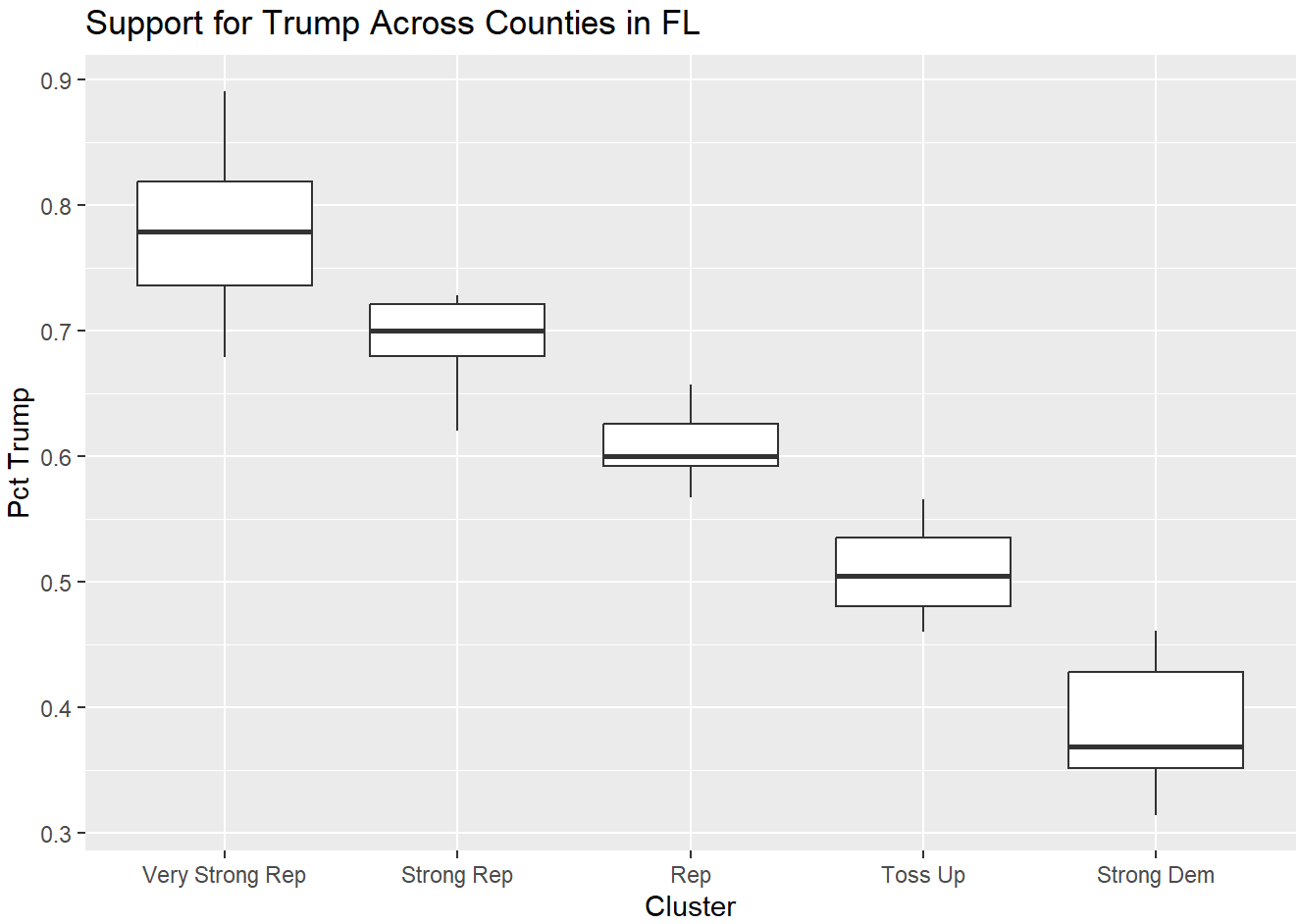
```
table(dat.cluster$cluster,dat.cluster$party.strata)
```

```
##
##                   1--High Democrat 2--Mod Democrat 3--Middle 4--Mod Republican
##   Very Strong Rep                 0                0         0                  0
##   Strong Rep                      0                0         0                  0
##   Rep                             0                0         0                  9
##   Toss Up                         0                0         7                  2
##   Strong Dem                      3                5         0                  0
##
##                   5--High Republican
##   Very Strong Rep                 19
##   Strong Rep                      14
##   Rep                              8
##   Toss Up                          0
##   Strong Dem                       0
```

Interesting….

It is often also useful to summarize the distribution of key variables by the clusters we have found to try to interpret their meaning. Here we can use a boxplot to describe how the county clusters vary in terms of the average support for President Trump.

```
dat.cluster %>%
  ggplot(aes(x=cluster, y=pct_rep_2020)) +
  geom_boxplot() +
  labs(x="Cluster",
       y="Pct Trump",
       title="Support for Trump Across Counties in FL")
```




We could also merge in county-level demographic data (using Census `fips_code`) and see how things change if we cluster counties based on their demographic features. But the important thing to remember is that because this is an unsupervised method there is no way to determine if the clustering is what you want it to be. Also recall that the scale matters. The computer will always find the number of clusters you ask for, but whether those clusters mean anything is up to you, the data analyst to determine. This is where critical thinking is essential – what variables are appropriate to include? And how do you interpret the meaning of the clusters given the distribution of data within those clusters?

# Even More Data! Even More Clusters ?

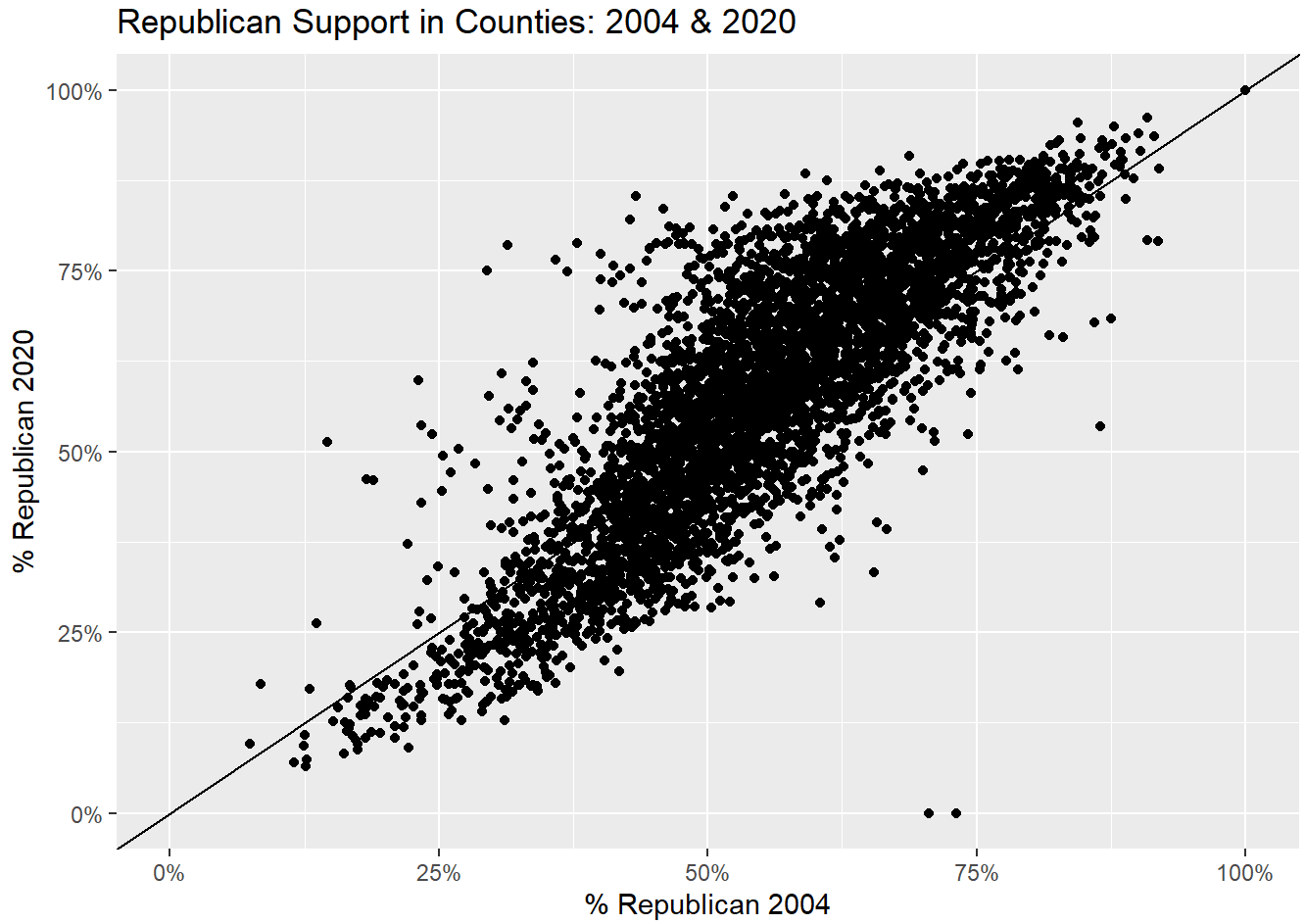
What if we looked at all counties? Note we are going to drop some states that do not record vote by counties (e.g., Maine) as well as others for which we are lacking data for some years (e.g., Alaska). Let's create a tibble containing just the data called `rawdata` and drop all missing data.

Do we need to standardize? Why or why not?

```
rawvote <- dat.all %>%
  select(c(pct_rep_2004,pct_rep_2008,pct_rep_2012,pct_rep_2016,pct_rep_2020)) %>%
  drop_na()
```

What if we compare Republican vote share in 2004 and 2020. What does that show? Let's plot and see.

```
rawvote %>%
  ggplot(aes(x=pct_rep_2004, y=pct_rep_2020)) +
  geom_point() +
  labs(x="% Republican 2004",
       y = "% Republican 2020",
       title = "Republican Support in Counties: 2004 & 2020") +
  geom_abline(intercept=0,slope=1) +
  scale_x_continuous(limits = c(0,1),labels = scales::percent_format(accuracy = 1)) +
  scale_y_continuous(limits = c(0,1),labels = scales::percent_format(accuracy = 1))
```




We begin by setting a seed to ensure replicability and then we fit `K` different clusters – one for each choice of `K`.

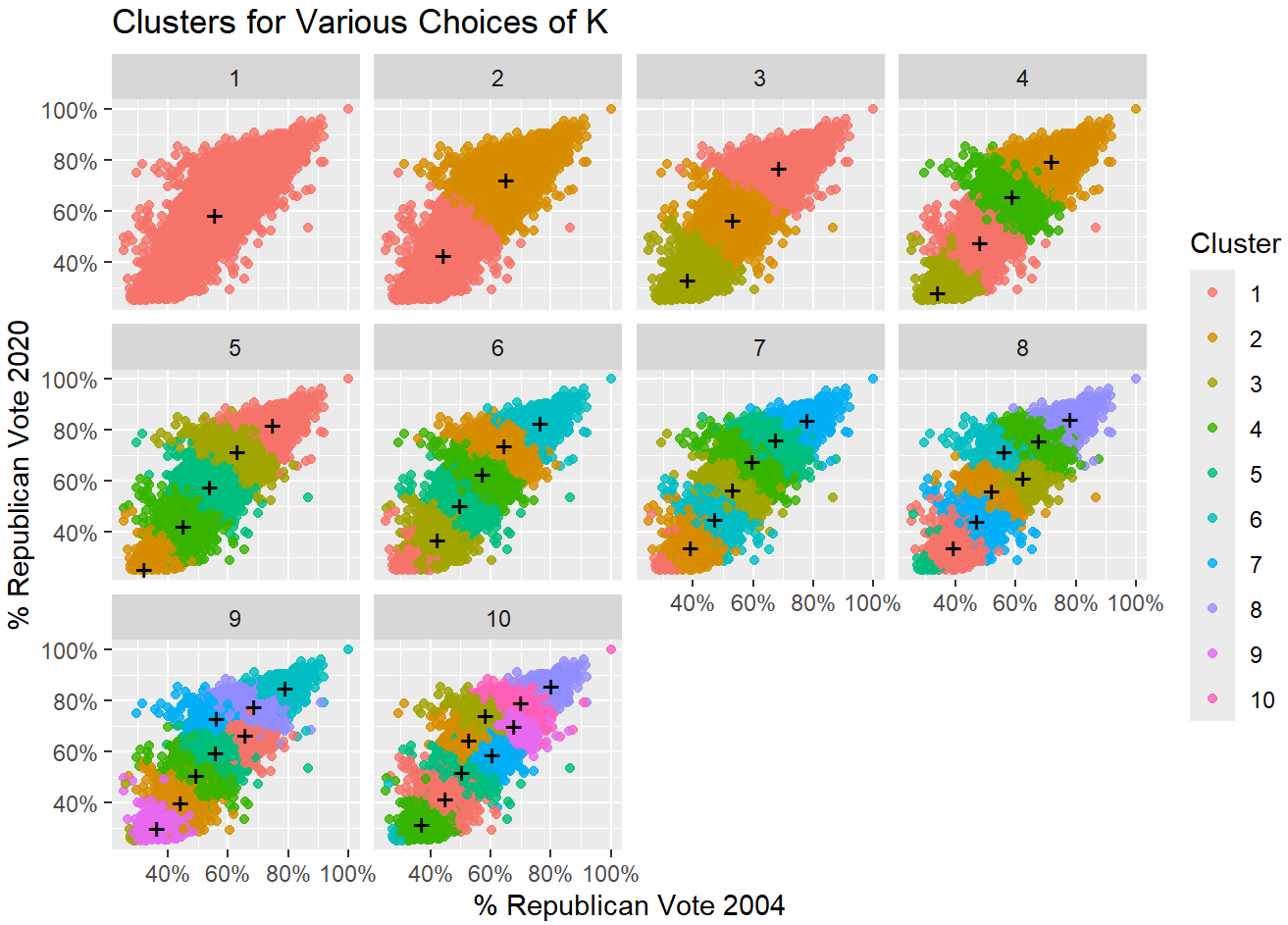
```
set.seed(42)
kcluster.us <-
  tibble(K = 1:10) %>%   # define a sequence that we will use to denote k
  mutate(   # now we are going to create new variables in this tibble
    kcluster = map(K, ~kmeans(rawvote, .x, iter.max = 100)),   # run a kmeans clustering using k
    tidysummary = map(kcluster, tidy), # run the tidy() function on the kcluster object
    augmented = map(kcluster, augment, rawvote) # save the cluster to the data
  )
```

To plot this we want to extract the data points from `kcluster.us` using the `unnest` function to the tibble `points.us` and we want to extract the centroids of the clusters from the tidysummary for each cluster into the new tibble `clusters.us` by `unnest`ing the summaries of each fit.

```
points.us <- kcluster.us %>%
  unnest(cols=c(augmented))

clusters.us <- kcluster.us %>%
  unnest(cols=c(tidysummary))
```

Now we can use these two new tibbles to plot.

```
points.us %>%
  ggplot(aes(x = pct_rep_2004, y = pct_rep_2020)) +
  geom_point(aes(color = .cluster), alpha = 0.8) +
  labs(x = "% Republican Vote 2004",
       y = "% Republican Vote 2020",
       color = "Cluster",
       title = "Clusters for Various Choices of K") +
  facet_wrap(~ K) +
  scale_x_continuous(limits = c(.25,1),labels = scales::percent_format(accuracy = 1)) +
  scale_y_continuous(limits = c(.25,1),labels = scales::percent_format(accuracy = 1)) +
  geom_point(data = clusters.us, size = 4, shape = "+")
```



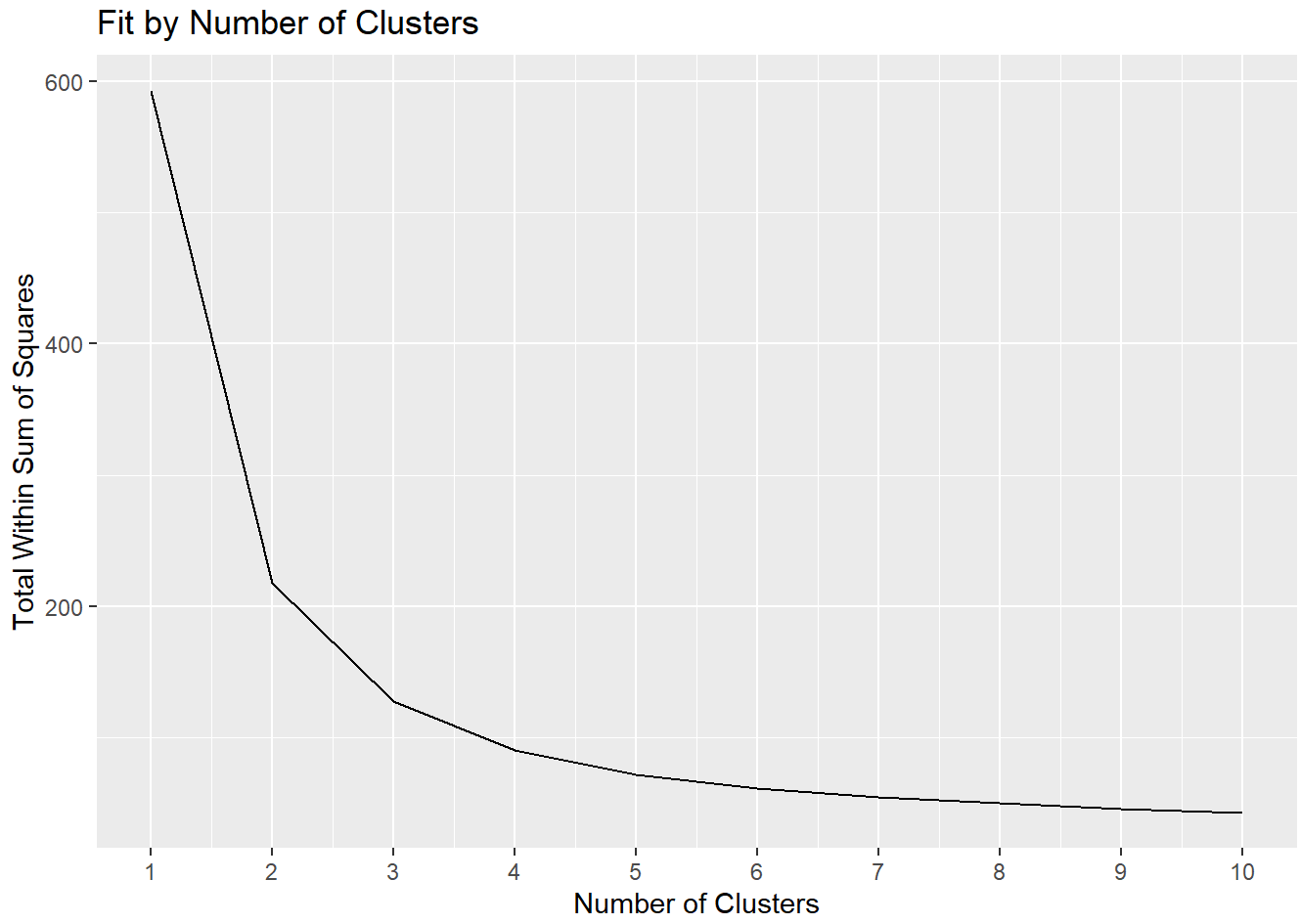

So how many clusters? Here we can see what the total within sum of squares is for each set of clusters that we find for the various choices of k. To determine how many, we want to choose a value of k such that there is very little change from adding additional clusters. Note that more clusters will always do better, so the issue is to find out the point at which the improvement seems small. This is a judgment call.

So let's extract the fits from the `kcluster.us` list and then loop over the `k` different fits to extract the total within sum of squares ( `tot.withinss` ) and then create a new tibble `fit` that contains the vector of cluster sizes and vector of total within sum of squares that we used the loop to extract (i.e., `tot.withinss` ).

```
fits <- kcluster.us[[2]]
tot.withinss <- NULL

for(i in 1:10){
  tot.withinss[i] <- fits[[i]]$tot.withinss
}

fit <- bind_cols(k = seq(1,10), tot.withinss = tot.withinss)
```

Now plot to see where the line "bends".

```
fit %>%
  ggplot(aes(x=k,y=tot.withinss)) +
  geom_line() +
  scale_x_continuous(breaks=seq(1,10)) +
  labs(x="Number of Clusters",
       y ="Total Within Sum of Squares",
       title = "Fit by Number of Clusters")
```




So it seems like there are 4 or maybe 5 clusters that seem relevant?