Lab 8

Sakib Salim

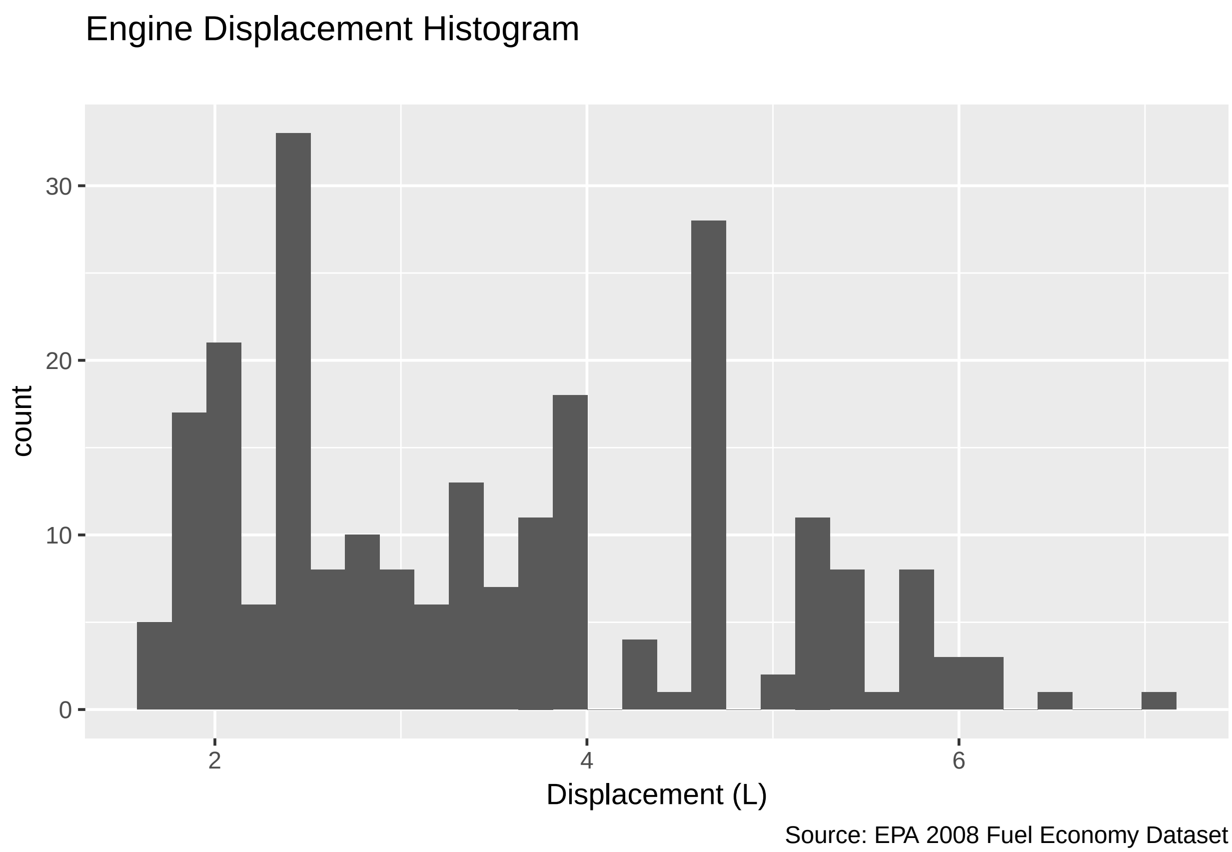
Load the `ggplot2` library and its dataset called `mpg`. Print out a summary of the dataset using `summary` and `str`.

```
pacman::p_load(ggplot2, dplyr, stringr, Rcpp, ggExtra, ggcorrplot, devtools)
data(mpg)
mpg$drv = factor(mpg$drv)
mpg$manufacturer = factor(mpg$manufacturer)
mpg$model = factor(mpg$model)
mpg$fl = factor(mpg$fl)
mpg$class = factor(mpg$class)
mpg$cyl = factor(mpg$cyl)
```

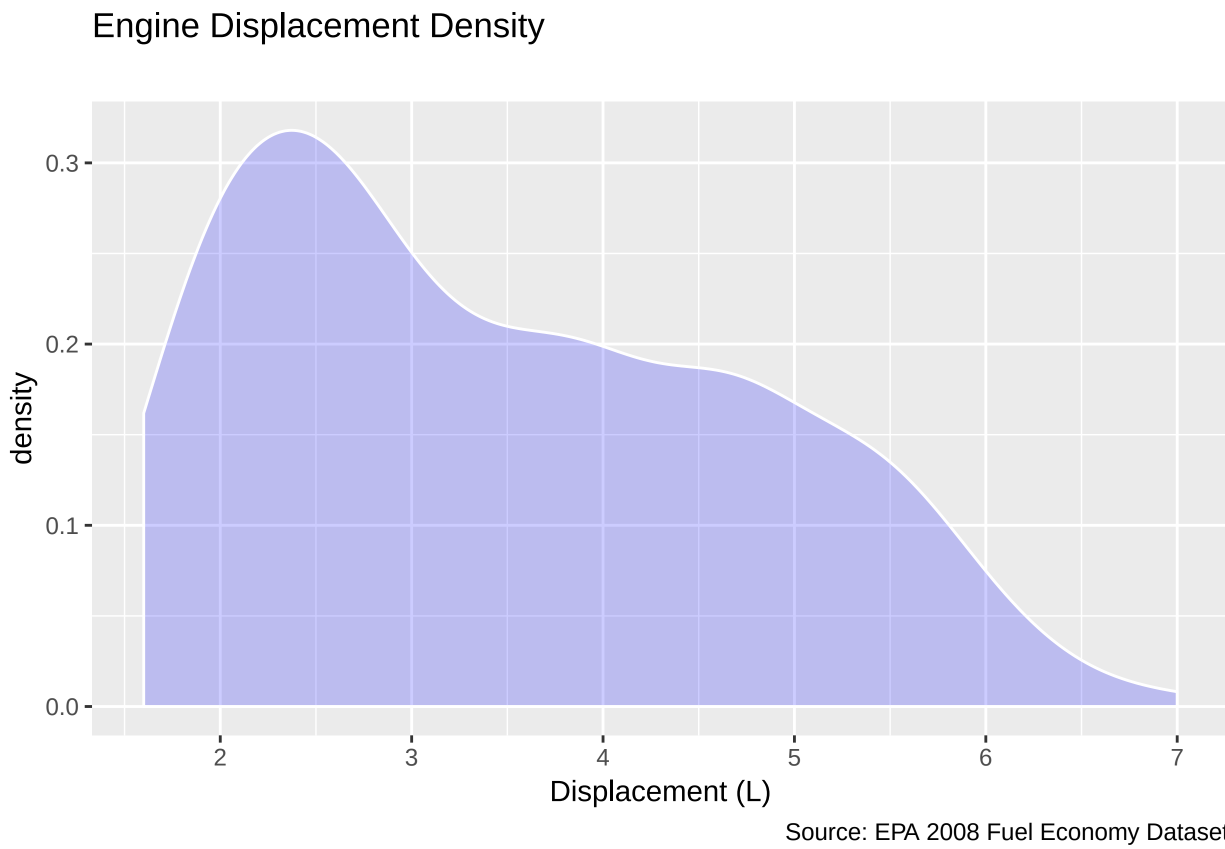
Visualize a histogram then a density estimate of the `displ` variable, the engine displacement. Use `labs` to create a `title`, `subtitle`, `caption` and x-label via `x` and y-label via `y`. Do this for every single illustration in this lab.

```
ggplot(mpg) +
  aes(displ) +
  geom_histogram() +
  labs(title = "Engine Displacement Histogram" , subtitle = "", x = "Displacement (L)", caption = "Sour
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



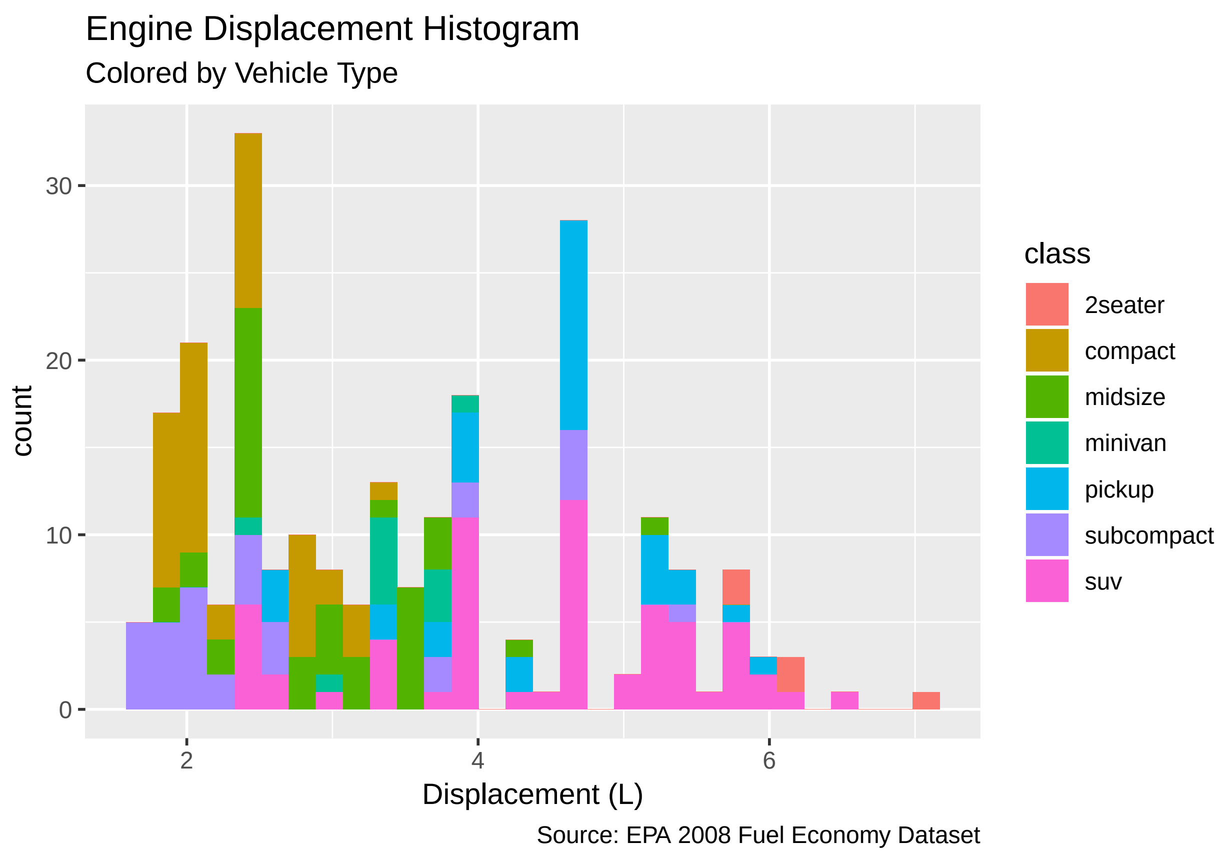
```r
ggplot(mpg) +
  aes(displ) +
  geom_density(fill = "blue", alpha = 0.2, col = "white") +
  labs(title = "Engine Displacement Density" , subtitle = "", x = "Displacement (L)", caption = "Source
```



Visualize a histogram the `displ` variable, but then fill the color of the bar by the `class` of the car. You will have to pass `class` in as the `fill` in the aesthetic of the histogram.

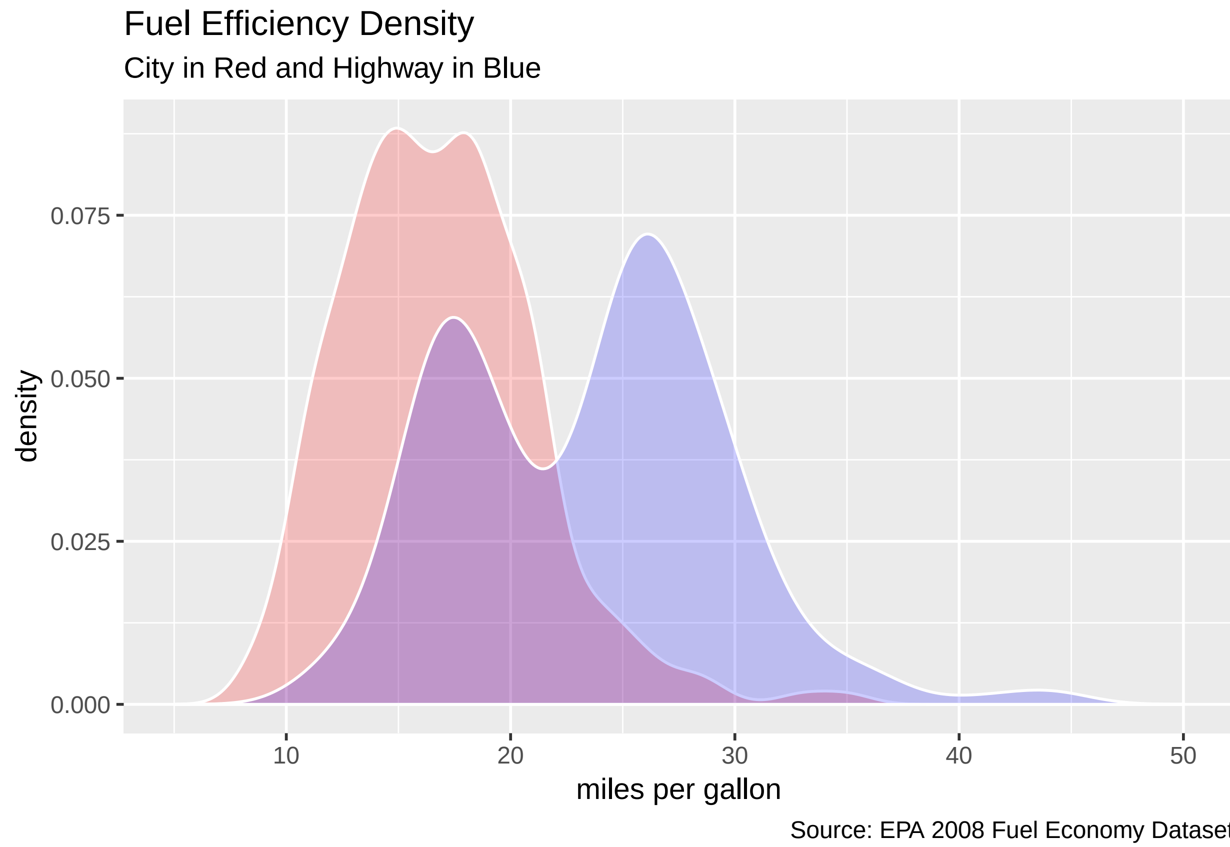
```r
ggplot(mpg) +
  aes(displ) +
  geom_histogram(aes(fill = class)) +
  labs(title = "Engine Displacement Histogram" , subtitle = "Colored by Vehicle Type", x = "Displacemen
```

```
## ‘stat_bin()‘ using ‘bins = 30‘. Pick better value with ‘binwidth‘.
```



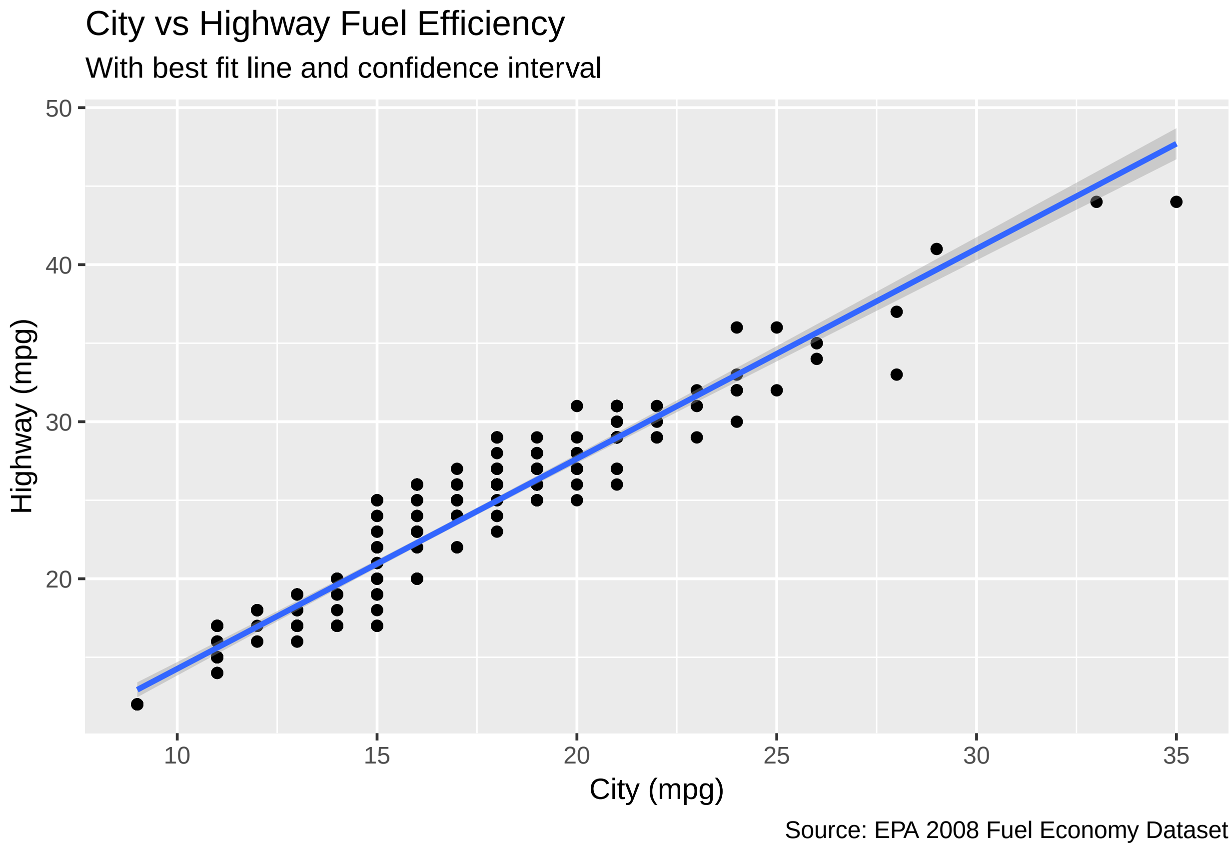
Visualize overlapping densities of `cty` (city miles per gallon) and `hwy` (highway miles per gallon) using two colors with an alpha blend.

```
ggplot(mpg) +
  geom_density(aes(cty), fill = "red", col = "white", alpha = 0.2) +
  geom_density(aes(hwy), fill = "blue", col = "white", alpha = 0.2) +
  xlim(5, 50) +
  labs(title = "Fuel Efficiency Density" , subtitle = "City in Red and Highway in Blue", x = "miles per
```



Plot `cty` (city miles per gallon) vs `hwy` (highway miles per gallon) and draw a best fit line with a confidence region of that line.

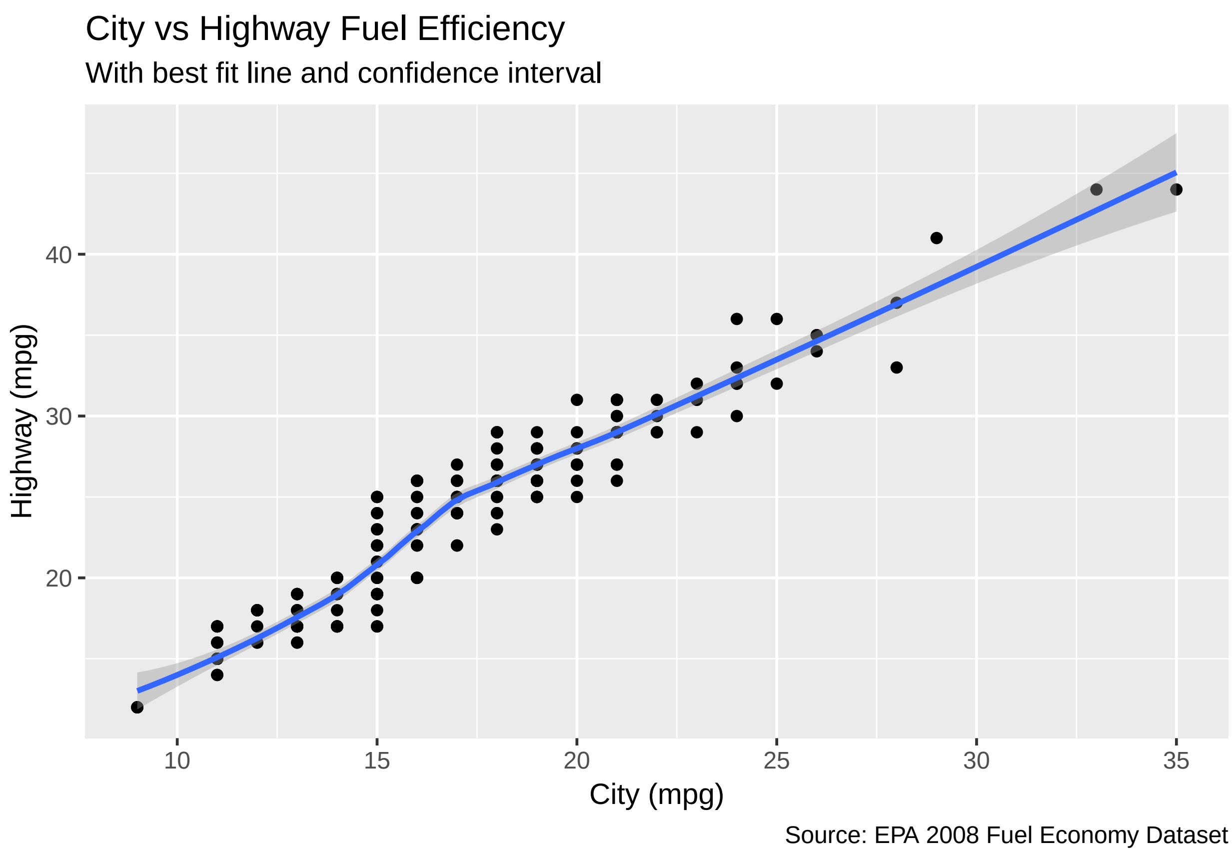
```
ggplot(mpg, aes(x = cty, y = hwy)) +
  geom_point() +
  geom_smooth(method = "lm") +
  labs(title = "City vs Highway Fuel Efficiency", subtitle = "With best fit line and confidence interva
```



Plot `cty` (city miles per gallon) vs `hwy` (highway miles per gallon) and draw a best fit non-parametric functional relationship with a confidence region of that relationship.

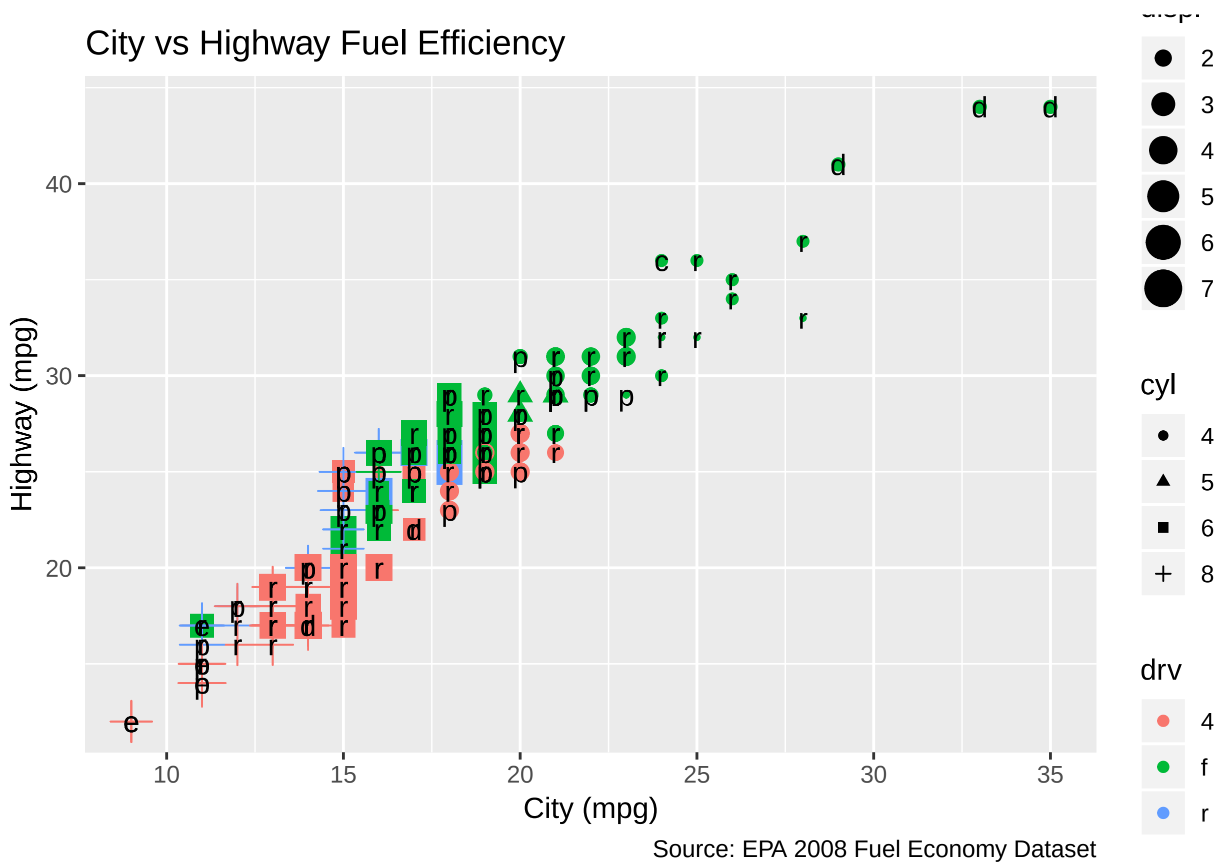
```
ggplot(mpg, aes(x = cty, y = hwy)) +
  geom_point() +
  geom_smooth() +
  labs(title = "City vs Highway Fuel Efficiency", subtitle = "With best fit line and confidence interval
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



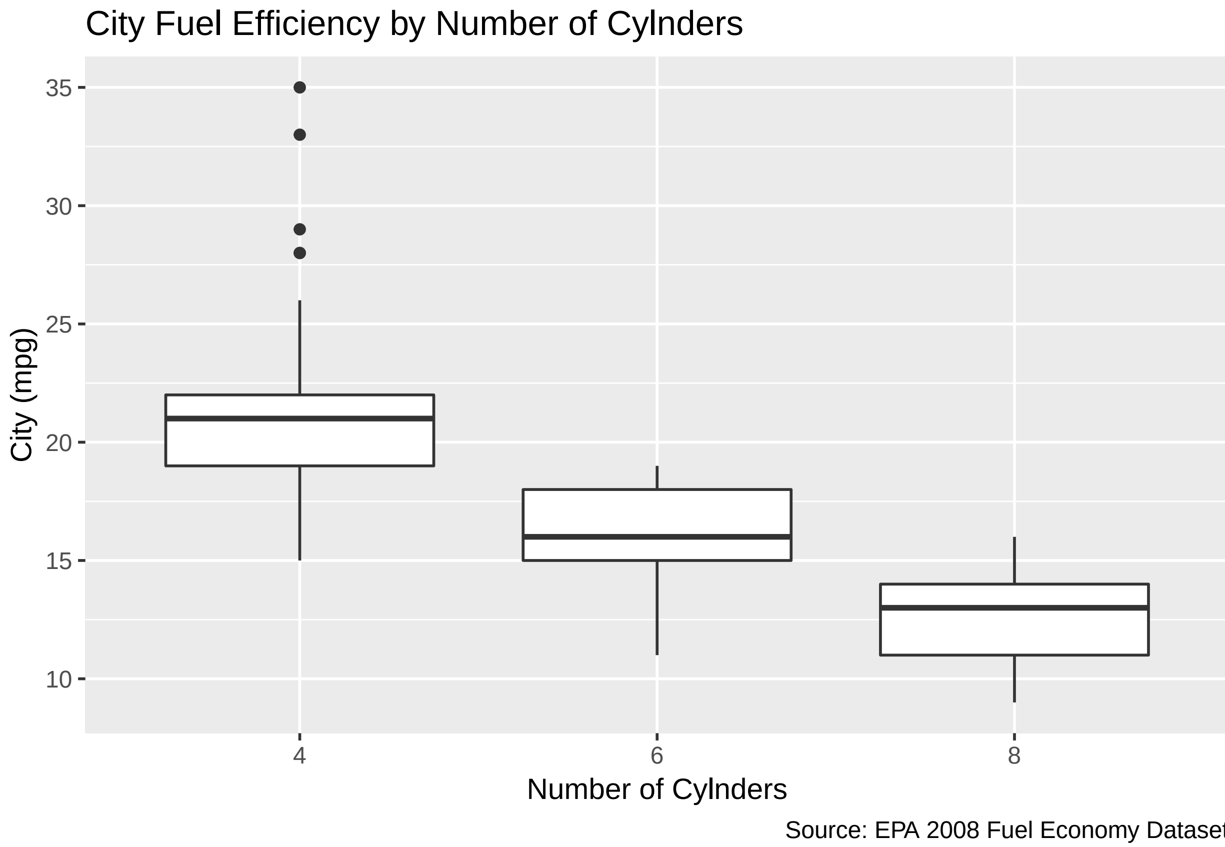
Plot `cty` (city miles per gallon) vs `hwy` (highway miles per gallon) and then try to visualize *as many other variables* as you can visualize effectively on the same plot. Try text, color, size, shape, etc.

```
ggplot(mpg, aes(x = cty, y = hwy)) +
  geom_point(aes(col = drv, shape = cyl, size = displ)) +
  geom_text(aes(label = fl)) +
  labs(title = "City vs Highway Fuel Efficiency", x = "City (mpg)", y = "Highway (mpg)", caption = "Sou
```



Convert `cyl` to an ordinal factor. Then use the package `dplyr` to retain only cars with 4, 6, 8 cylinders in the dataset. Then make a canonical illustration of `cty` by `cyl`.

```
mpg$cyl = factor(mpg$cyl, ordered = TRUE)
mpg = mpg %>%
  filter(cyl %in% c(4, 6, 8))
ggplot(mpg, aes(x = cyl, y = cty)) +
  geom_boxplot() +
  labs(title = "City Fuel Efficiency by Number of Cylnders", x = "Number of Cylnders", y = "City (mpg)"
```

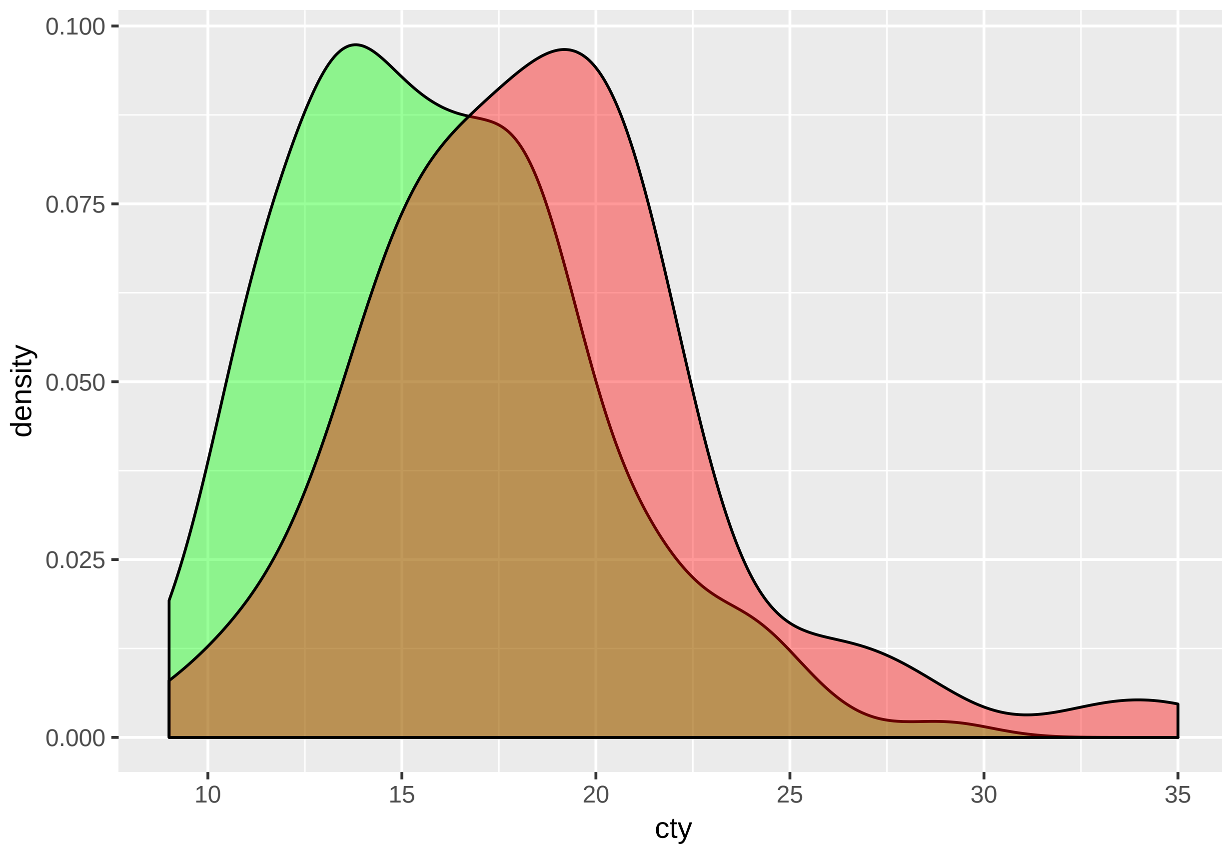


Load the `stringr` library. Use the `str_detect` function in this libary to rewrite the `trans` variable in the data frame to be just "manual" or "automatic".

```
#TO-DO
mpg$trans[str_detect(mpg$trans, "manual")] = "manual"
mpg$trans[str_detect(mpg$trans, "auto")] = "auto"
```

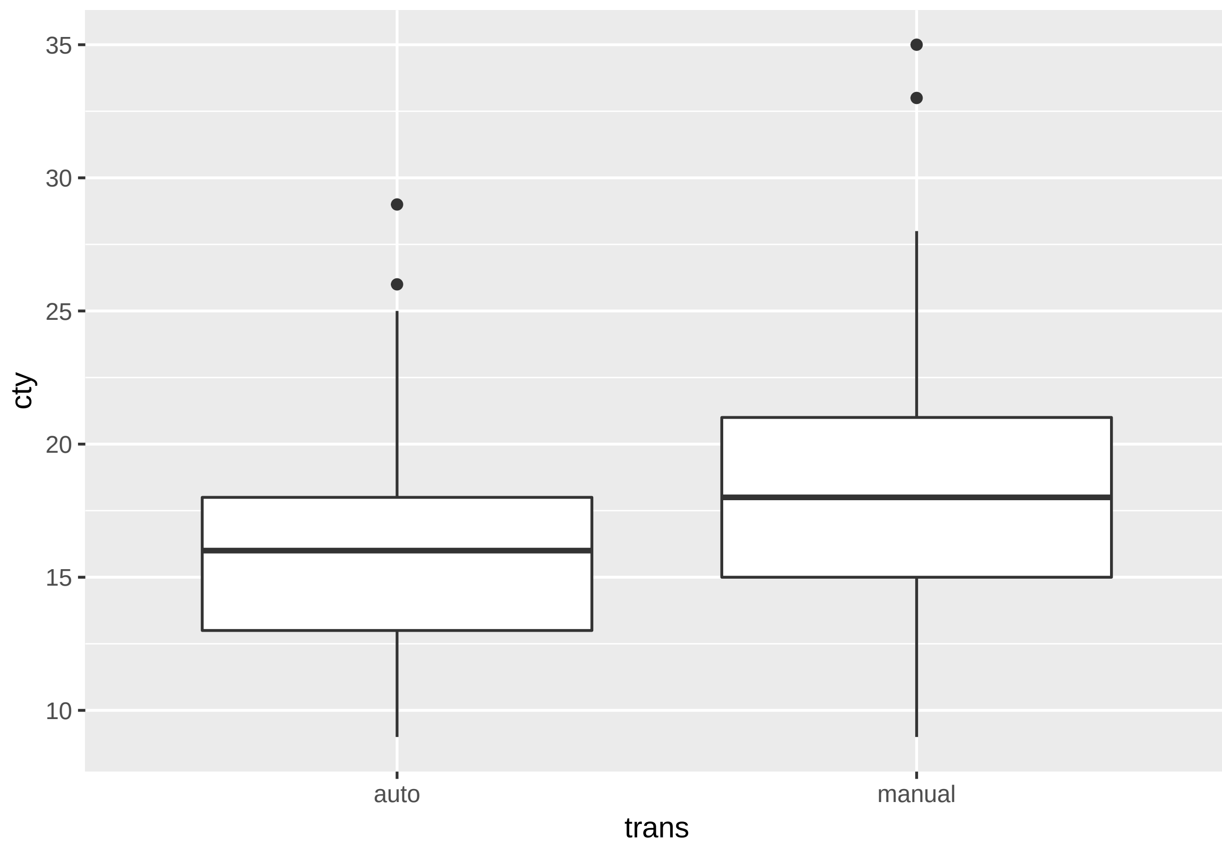
Now visualize `cty` by `trans` via two overlapping alpha-blended densities.

```
#TO-DO
ggplot(mpg, aes(cty)) +
  geom_density(data = subset(mpg, trans == "auto"), fill = "green", alpha = 0.4) +
  geom_density(data = subset(mpg, trans == "manual"), fill = "red", alpha = 0.4)
```



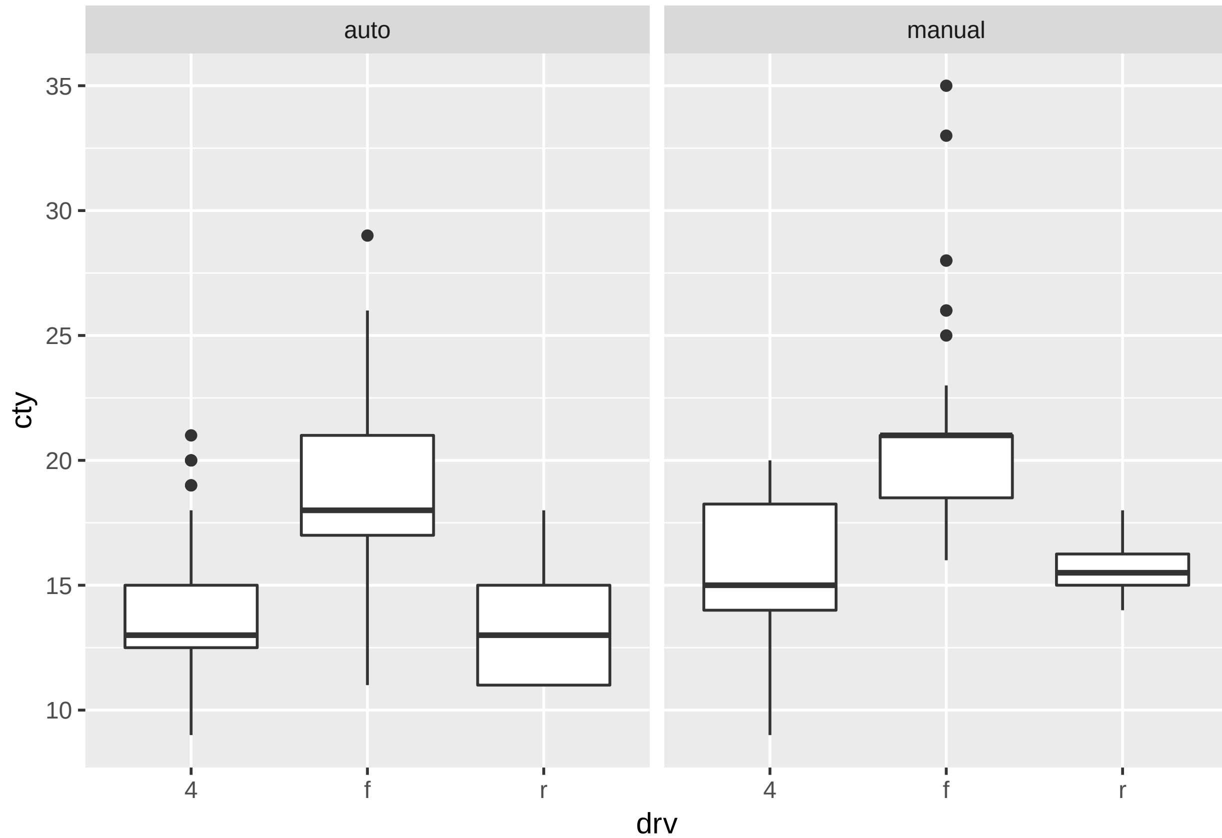
Now visualize `cty` by `trans` via a box and whisker plot.

```
#TO-DO
ggplot(mpg, aes(y = cty, x = trans)) +
  geom_boxplot()
```



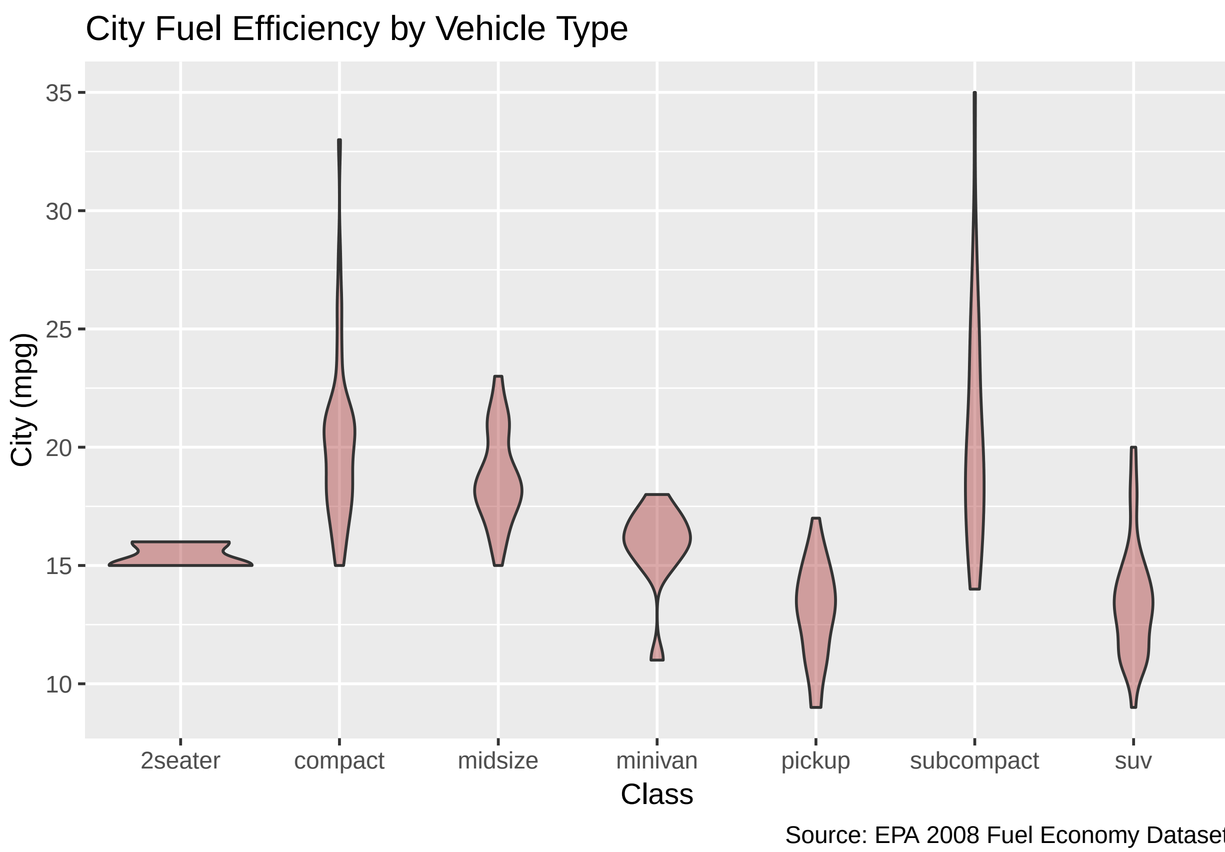
Now visualize `cty` by `drv` by `trans` via two box and whisker plots horizontally laid out.

```
#TO-DO
ggplot(mpg, aes(y = cty, x = drv)) +
  geom_boxplot() +
  facet_grid(. ~ trans)
```



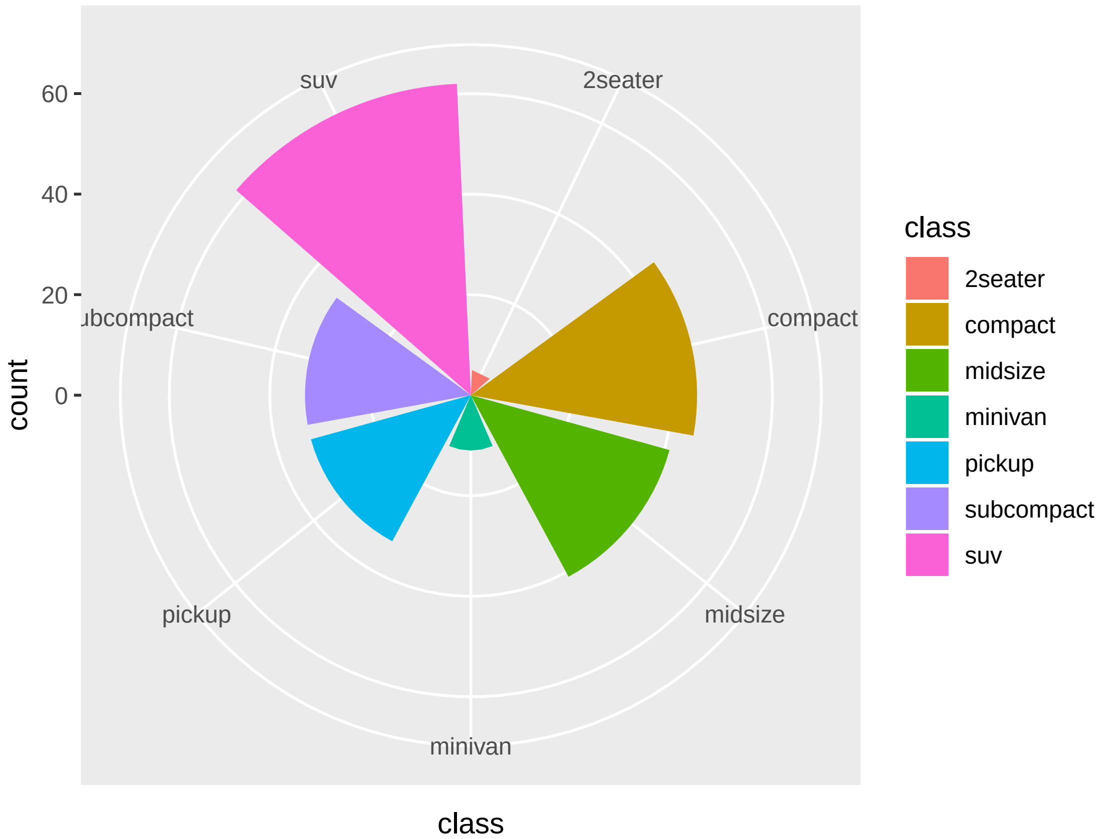
Now visualize `cty` by `class` via a violin plot. Look at the ggplot cheatsheet!

```
ggplot(mpg, aes(x = class, y = cty)) +
  geom_violin(fill = "brown", alpha = 0.4) +
  labs(title = "City Fuel Efficiency by Vehicle Type", x = "Class", y = "City (mpg)", caption = "Source
```



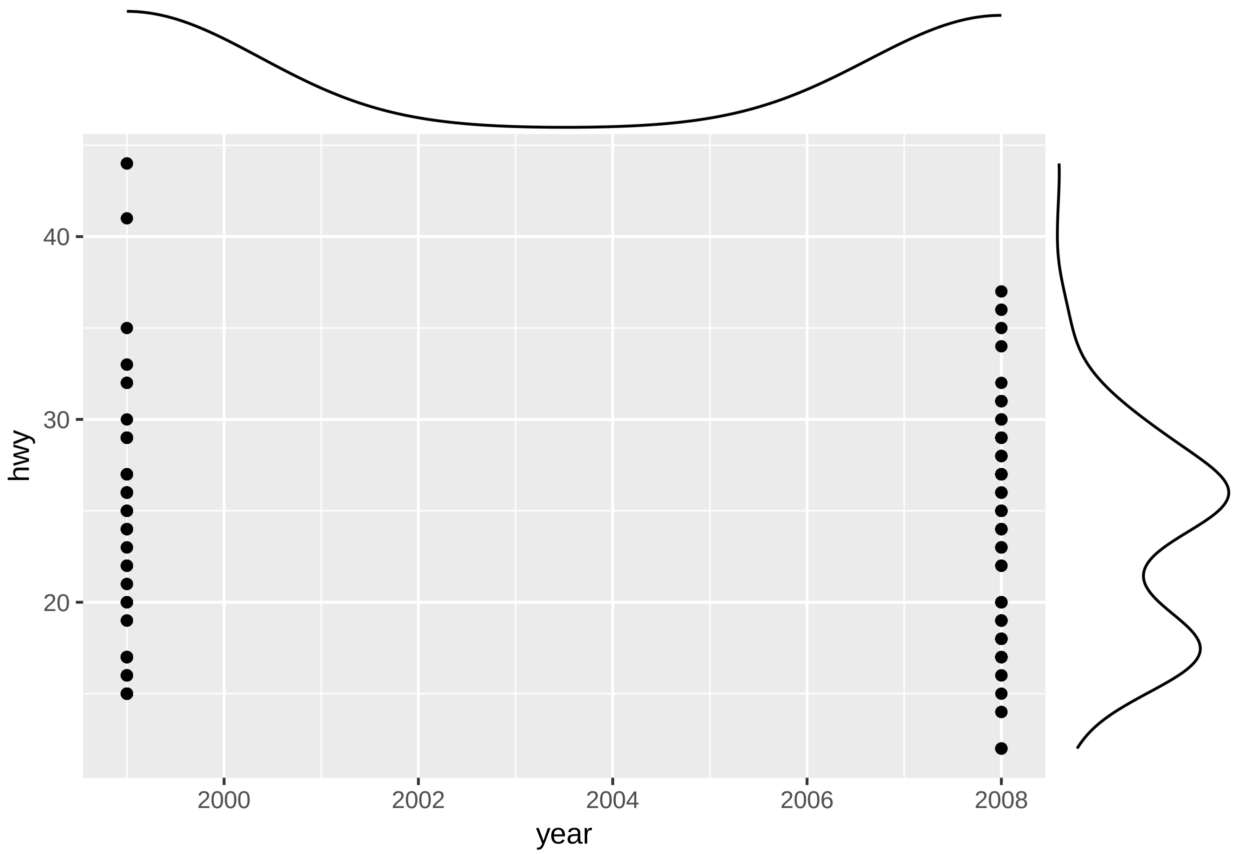
Make a pie chart of `class`. Visualize `trans` vs `class`. Look at the ggplot cheatsheet!

```
#TO-DO
ggplot(mpg, aes(class)) +
  geom_bar(aes(fill = class)) +
  coord_polar("x")
```



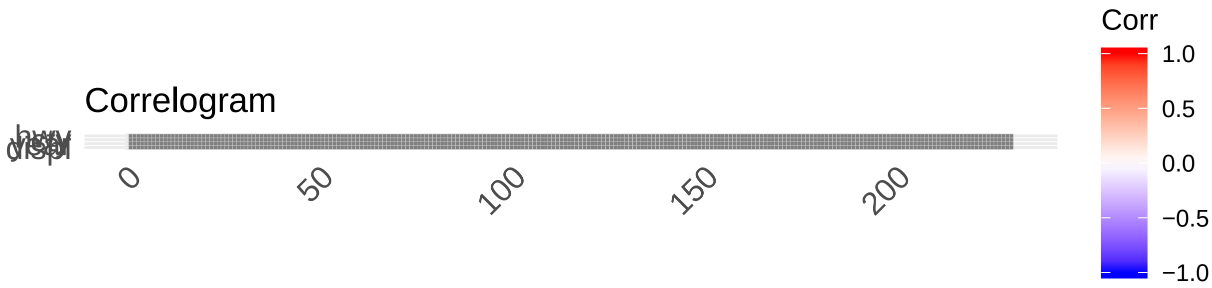
Using the package ggExtra's ggMarginal function, look at the hwy by year and plot the marginal density on both the x and y axes.

```
#TO-DO
ggMarginal(data = mpg, x = "year", y = "hwy", type = "density", margins = "both")
```



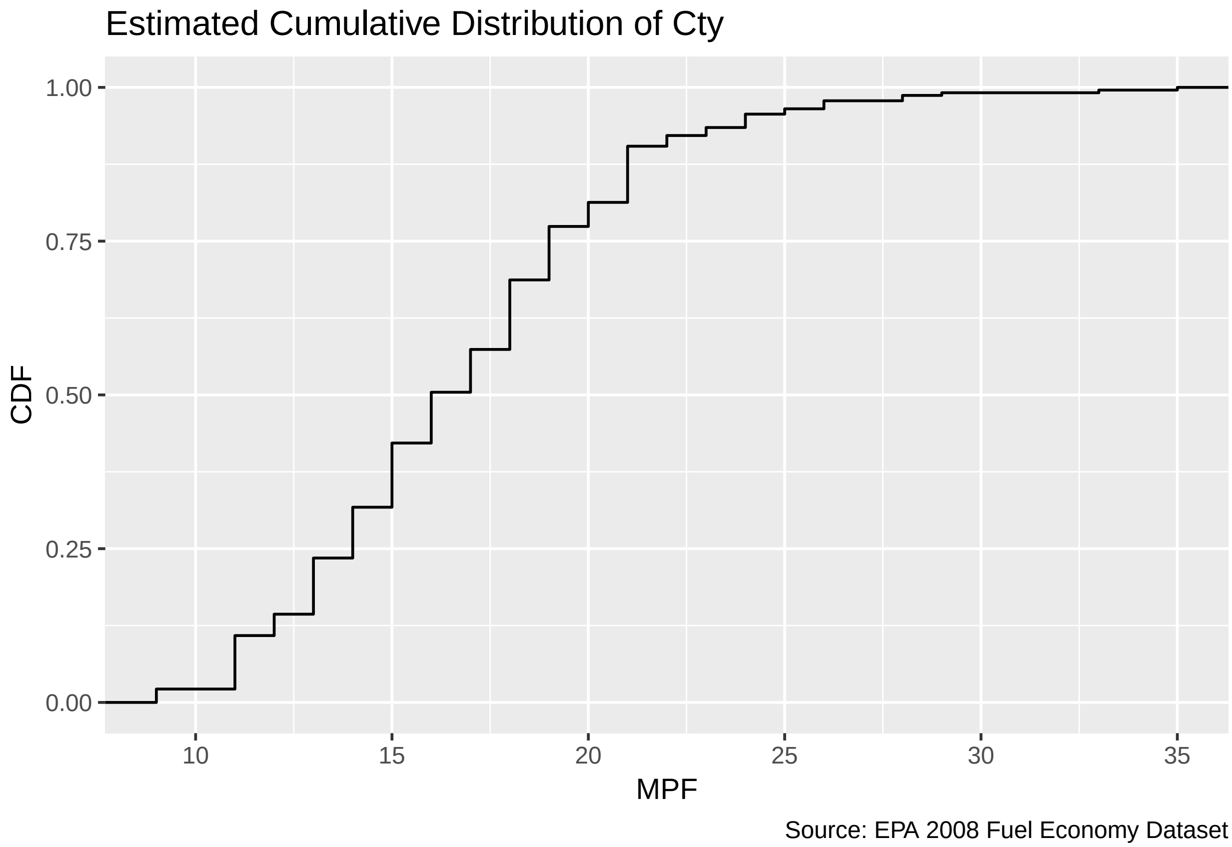
Using the package `ggcorrplot`'s `ggcorrplot` function, look at the correlations for all variables in this dataset that are legal in a corrrelogram. Use dplyr to `select_if` the variable is appropriate.

```
#TO-DO
mpg %>%
  select_if(is.numeric) %>%
  ggcorrplot(title = "Correlogram", legend.title = "Corr")
```



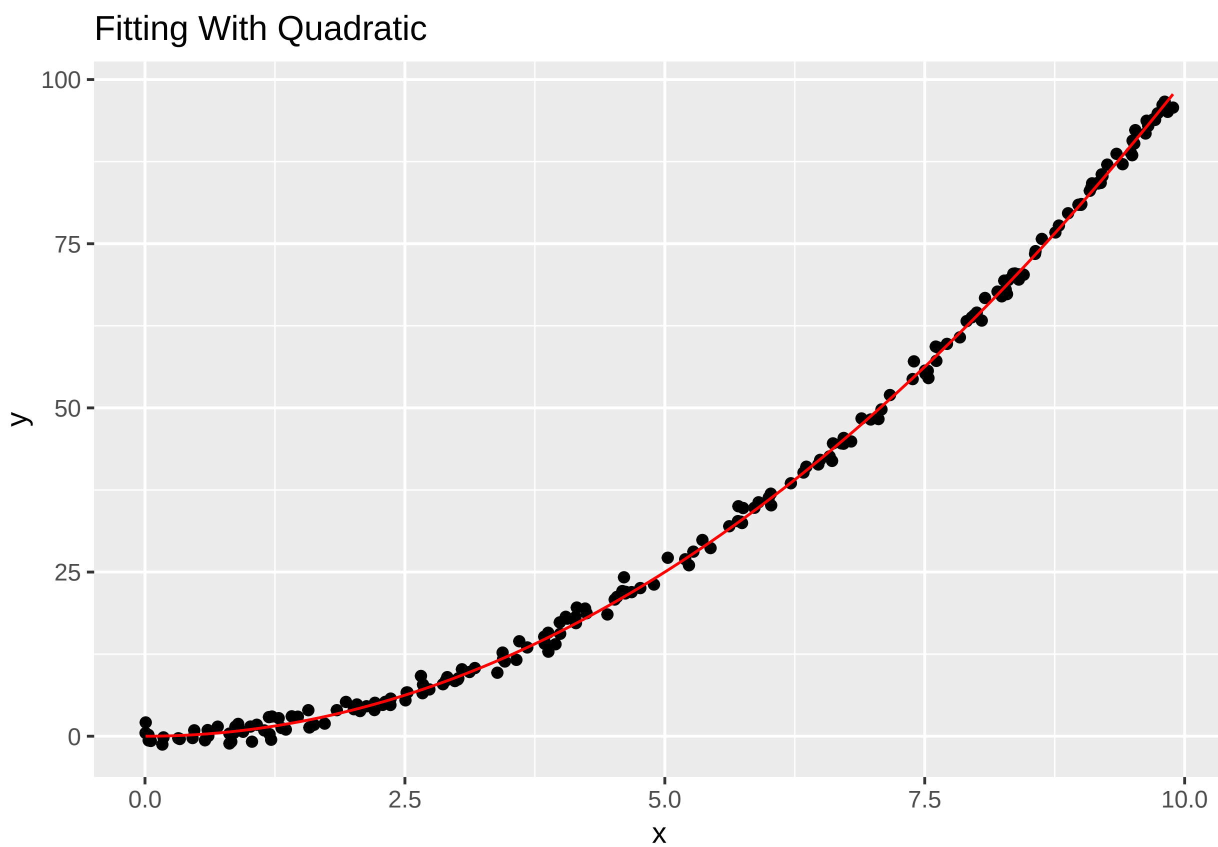
Use the `stat_ecdf` function to plot the estimated cumulative distribution of 'cty'.

```
#TO-DO
ggplot(mpg, aes(cty)) +
  stat_ecdf() +
  labs(title = "Estimated Cumulative Distribution of Cty", x = "MPF", y = "CDF", caption = "Source: EPA
```



Create a data generating process where x is uniform between 0 and 10 and y is x^2 plus $N(0, 1)$ noise. Plot $n = 200$ points and then plot the quadratic relationship $y = x^2$ using the function `stat_function`.

```r
#TO-DO
n = 200
df = data.frame (x = runif(n, 0, 10))
x = df$x
y = x^2 + rnorm(n)
ggplot(df, aes(x, y)) +
  geom_point() +
  stat_function(fun = function(x){x^2}, color = "red") +
  labs(title = "Fitting With Quadratic")
```



We now move to Rcpp. Load the library.

```
#TO-DO
```

Write an R function `is_odd` and a C++ function `is_odd_cpp` that evaluates if a number is odd and returns true if so.

```
#TO-DO
is_odd = function(n){
  return(n %% 2 == 1)
}

cppFunction('
  bool is_odd_cpp(int n){
    return (n % 2 == 1);
  }
')
```

Using 'system.time', run both functions 1,000,000 times on the numbers 1, 2, . . . , 1000000. Who is faster and by how much?

```
#TO-DO
Rcpp::evalCpp("2+2")
```

```
## [1] 4
```

```
system.time({
  for(i in 1:1e6)
    is_odd(i)
```

```
})
```

```
##    user  system elapsed
##    0.86    0.00    0.85
```

```
system.time({
  for(i in 1:1e6)
    is_odd_cpp(i)
})
```

```
##    user  system elapsed
##    2.25    0.00    2.25
```

```
#R takes about 1.07 seconds, and Rcpp takes about 2.80 seconds. So R is 162 percent faster.
```

Write an R function `fun` and a C++ function `fun_cpp` that takes a natural number returns n if n is 0 or 1 otherwise the result of the function on $n-1$ and $n-2$. This is the function that returns the nth Fibonacci number.

```
#TO-DO
fun = function(n){
  if(n == 0 || n == 1)
    return(n)
  return (fun(n-1) + fun(n-2))
}

cppFunction('
  int fun_cpp(int n){
    if (n == 0 || n == 1)
      return n;
    return fun_cpp(n-1) + fun_cpp(n-2);
  }
')
```

Using 'system.time', run both functions on the numbers 1, 2, . . . , 100. Who is faster and by how much?

```
#TO-DO
system.time({
  for(i in 1:25)
    fun(i)
})
```

```
##    user  system elapsed
##    0.54    0.00    0.53
```

```
system.time({
  for(i in 1:25)
    fun_cpp(i)
})
```

```
##    user  system elapsed
##       0       0       0
```

```
#Rcpp completes in 0.02 seconds while R takes 0.67 seconds
```

Write an R function `logs` and a C++ function `logs_cpp` that takes a natural number n and returns an array of $ln(1), ln(2), ..., ln(n)$.

```r
#TO-DO
logs = function(n){
  array(log(1:n))
}

cppFunction('
  NumericVector logs_cpp(int n){
    NumericVector ans(n);
    for (int i = 1; i <= n; i++){
      ans[i-1] = log(i);
    }
    return ans;
  }
')
```

Using 'system.time', run both functions on the numbers 1, 2, . . . , 1000000. Who is faster and by how much?

```r
#TO-DO
system.time({
  for(i in 1:1e4)
    logs(i)
})
```

```
##    user  system elapsed
##    2.71    0.11    2.92
```

```r
system.time({
  for(i in 1:1e4)
    logs_cpp(i)
})
```

```
##    user  system elapsed
##    2.27    0.00    2.26
```

```r
#Rcpp is slightly faster. It takes 2.25 seconds compared to 2.61 seconds, a 14 percent decrease in spee
```

Write an R function `max_distances` and a C++ function `max_distances_cpp` that takes an $n \times p$ matrix X and returns an $n \times n$ matrix called D of NA's where the upper triangular portion above the diagonal is the max distances between the elements of the i, jth rows of X.

```r
#TO-DO
max_distances = function(X){
  n = nrow(X)
  D = matrix(NA, n, n)
  for(i in 1:n){
    for (j in (i+1):n){
      if (j > n)
        j = n
      max_dist = 0
      for (k in 1:ncol(X)){
        ij_dist = abs(X[i,k] - X[j,k])
        if ( ij_dist > max_dist){
          max_dist = ij_dist
        }
      }
      D[i,j] = max_dist
    }
```

```
  }
  return(D)
}

cppFunction('
  NumericMatrix max_distances_cpp(NumericMatrix X){
    int n = X.nrow();
    int p = X.ncol();
    NumericMatrix D(n,n);
    for(int i = 0; i <= n; i++){
      for(int j = i+1; j <= n; j++){
        if ( j == n+1 )
          j = n;
        double max_dist = 0;
        for(int k = 0; k <= p; k++){
          double ij_dist = abs( X(i,k) - X(j,k) );
          if( ij_dist > max_dist  )
            max_dist = ij_dist;
        }
        D(i,j) = max_dist;
      }
    }
    return D;
  }
')
#Note: Indexing matrices starts at 0. nrow/ncol are methods.
```

Create a matrix X of $n = 1000$ and $p = 20$ filled with iid $N(0,1)$ realizations. Using 'system.time', calculate D using both functions. Who is faster and by how much?

```
#TO-DO
n = 1000
p = 20
X = matrix(rnorm(n*p), nrow = n, ncol = p)
system.time({
  max_distances(X)
})
```

```
##    user  system elapsed
##    3.94    0.00    3.93
```

```
system.time({
  max_distances_cpp(X)
})
```

```
##    user  system elapsed
##    0.03    0.00    0.03
```

```
#R takes 3.93 seconds while Rcpp takes 0.01 seconds, making Rcpp almost 400x faster!
```