Machine Learning and Data Visualization with the King County Housing Data

Lisa Huang

August - October 2018

This report is a summary of a project I completed in which I built statistical machine learning models to predict housing prices from a data set of houses located in King County, WA. The data were obtained from Kaggle at https://www.kaggle.com/harlfoxem/housesalesprediction. Each observation in the data set represents a house located in King County. The goal was to predict house prices based on features of the houses, including square footage, the number of bedrooms and bathrooms, and condition, among others. I also created graphical visualizations of the data and plotted relationships among the variables using ggplot in R. Some of these visualizations are presented here.

The models that I built to predict housing prices come from supervised learning methods for regression and classification, as well as unsupervised learning methods. In this report, I present the code and analyses for the supervised regression methods, including subset selection for linear regression, ridge regression and lasso, principal components regression, and random forests. For the full scripts and analyses, please visit my github page at https://github.com/lisahuang2/housing_data_ML.

Contents

e. principal components regression (PCR) and partial least squares (PLS)
f. random forests

6. Compare models

---

1. Import the data

First, import the data and view the dimensions of the data frame. You can see that there are 21613 observations in the data set, with each observation representing one house located in King County, Washington. You can view all the features in the data set using the names() function.

```
king <- read.csv("kc_house_data.csv")
dim(king)
```

```
## [1] 21613    21
```

```
names(king)
```

```
##  [1] "id"            "date"          "price"         "bedrooms"
##  [5] "bathrooms"     "sqft_living"   "sqft_lot"      "floors"
##  [9] "waterfront"    "view"          "condition"     "grade"
## [13] "sqft_above"    "sqft_basement" "yr_built"      "yr_renovated"
## [17] "zipcode"       "lat"           "long"          "sqft_living15"
## [21] "sqft_lot15"
```

---

2. Create a map of the houses in King County

First, we'll create a geographical map of the houses in King County, using the maps and ggplot2 libraries. The maps library is used to obtain a shape outline of King County in Washington, and the ggplot library is used for plotting the points on the map using latitude and longitude coordinates.

```
library(maps)
library(ggplot2)
```

To obtain the map outline of King County from the maps library, use the map_data() function to pull the county-level data from Washington state. Then subset the king county subregion.

```
washington <- map_data("county", region="washington")
king.county <- subset(washington, subregion=="king")
```

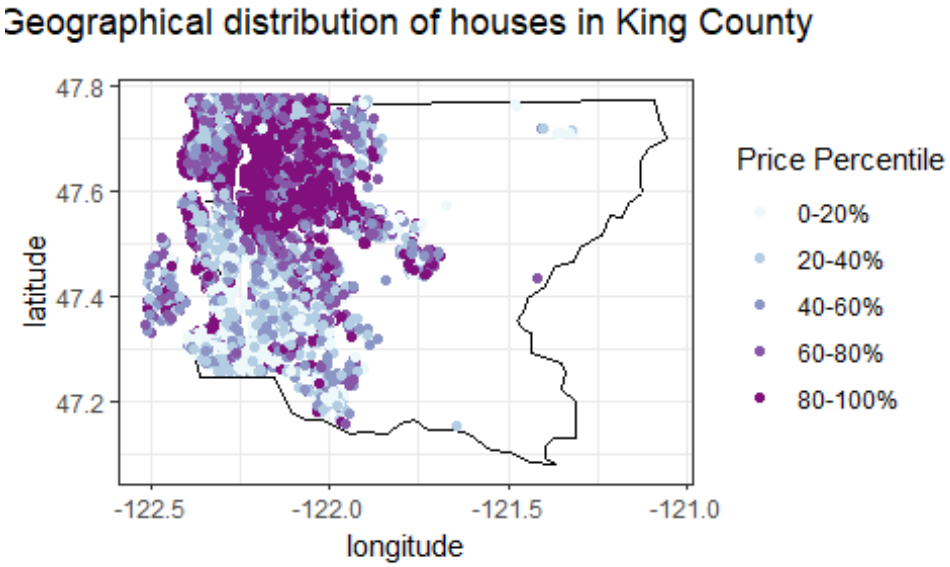
Before mapping the houses, cut the data into price quantiles. These quantiles will be mapped by color in order to visualize differences in housing prices across king county.

```
qa <- quantile(king$price, c(0, .2, .4, .6, .8, 1))
king$price_q <- cut(king$price, qa,
                    labels=c("0-20%", "20-40%", "40-60%", "60-80%", "80-100%"),
                    include.lowest=TRUE)
```

Finally, create the map with ggplot. Plot the king county shape outline from the maps package using geom_path(), then overlay the houses with geom_point().

```
library(ggplot2)
ggplot() +
  geom_path(data=king.county, aes(x=long, y=lat, group=group), color="black") +
```

```
  geom_point(data=king, aes(x = long, y = lat, color=price_q)) +
  scale_color_brewer(palette="BuPu") +
  labs(x="longitude", y="latitude", color="Price Percentile",
       title="Geographical distribution of houses in King County") +
  coord_quickmap() +
  theme_bw() +
  theme(plot.title=element_text(hjust=0.5, vjust=5))
```



As we can see from the map, almost all of the houses are concentrated on the west side of the county, with the most expensive houses in the northwest side, around Bellevue and Seattle.

---

3. Explore the features in the data set

There are several features in the data set which we will use to predict housing price. These features include the square footage of the house (sqft_living), square footage of the lot (sqft_lot), square footage of the living area measured in 2015 (sqft_living15), square footage of the lot measured in 2015 (sqft_lot15), the number of bedrooms in the house (bedrooms), the number of bathrooms in the house (bathrooms), the number of floors (floors), whether the house has a waterfront view (waterfront), the condition of the house (condition), the quality of the house (grade), and the year it was built (yr_built). Below are the summary statistics and distribution plots for a handful of these features.

First, open the ggplot2 library to plot the data:

```
library(ggplot2)
```

a. price (the price of each house)

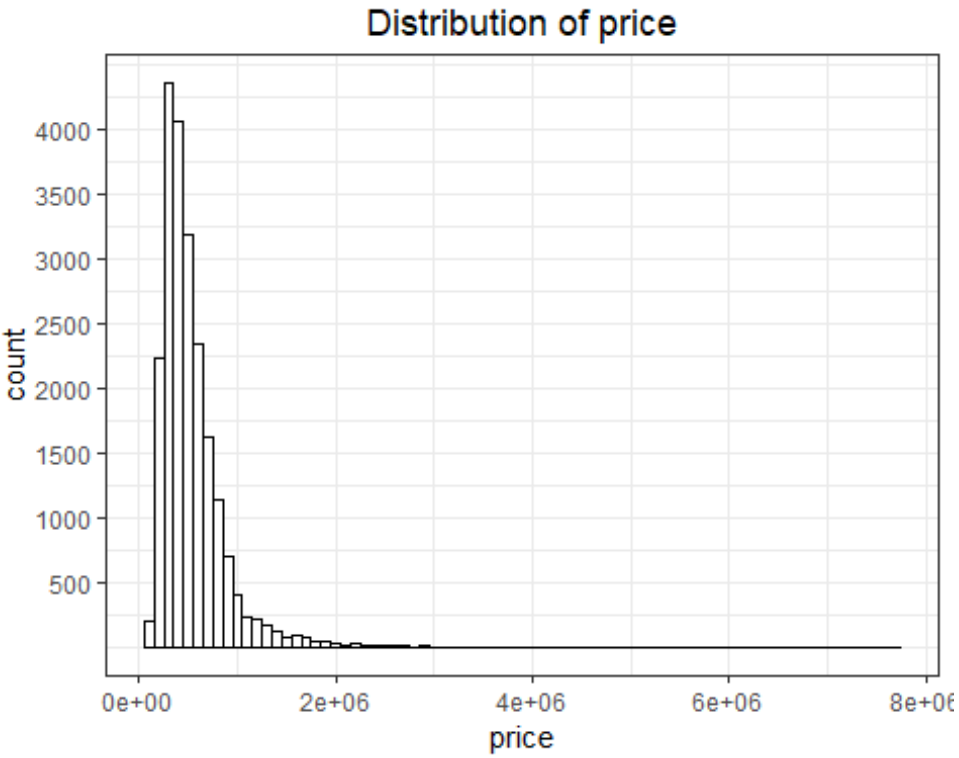
We first examine the price of each house. According to the summary below, the median house price is 450,000 dollars, and the mean is 540,088 dollars. The most expensive house is 7.7 million dollars.

```
summary(king$price)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   75000  321950  450000  540088  645000 7700000
```

We create a histogram using geom_histogram() in ggplot. You can see in the histogram that there is a strong positive skew in the distribution.

```
ggplot(king, aes(x=price)) +
  geom_histogram(binwidth=100000, color="black", fill="white") +
  scale_y_continuous(breaks=c(500, 1000, 1500, 2000, 2500, 3000, 3500, 4000)) +
  labs(title="Distribution of price") +
  theme_bw() +
  theme(plot.title=element_text(hjust=0.5))
```



b. sqrt_price (the square root of price)

Since price has a strong positive skew, let's create a new variable called sqrt_price, which is the square root transformation of price. From the summary below, the median of sqrt_price is 670.8, and the mean is 706.3.

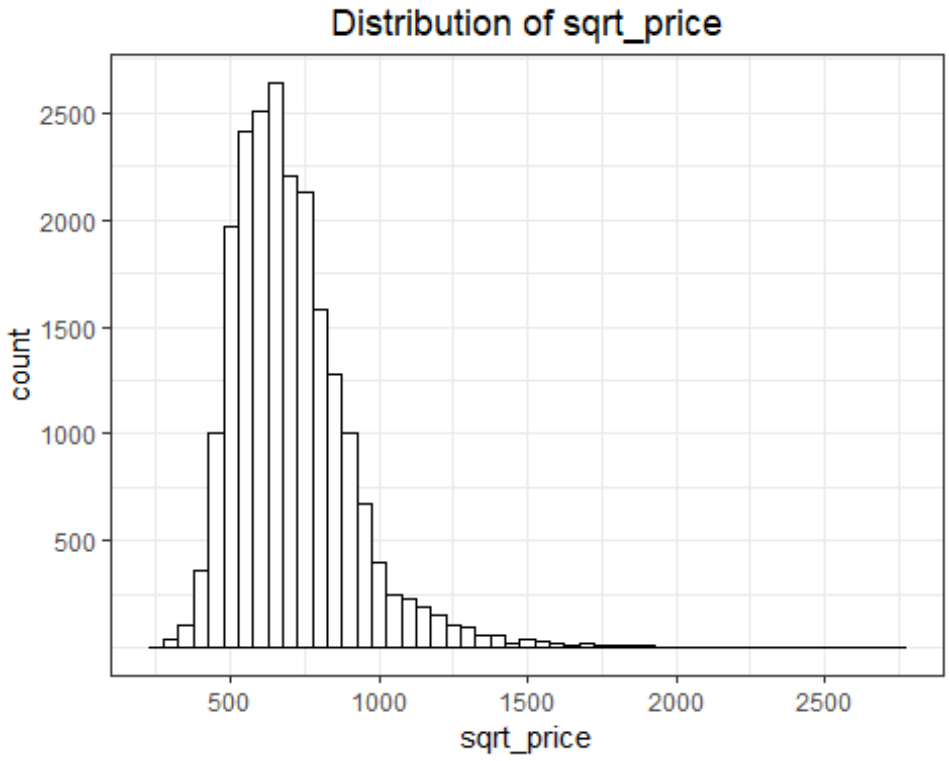
```
king$sqrt_price <- sqrt(king$price)
summary(king$sqrt_price)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   273.9   567.4   670.8   706.3   803.1  2774.9
```

From the histogram, we can see that the distribution of sqrt_price is still skewed, but it is less severe than in the original price variable. Thus, we will fit all of our statistical models on sqrt_price rather than on price.

```
ggplot(king, aes(x=sqrt_price)) +
  geom_histogram(binwidth=50, color="black", fill="white") +
  scale_x_continuous(breaks=c(500, 1000, 1500, 2000, 2500)) +
  scale_y_continuous(breaks=c(500, 1000, 1500, 2000, 2500)) +
```

```
  labs(title="Distribution of sqrt_price") +
  theme_bw() +
  theme(plot.title=element_text(hjust=0.5))
```



c. sqft_living (square footage of the house)

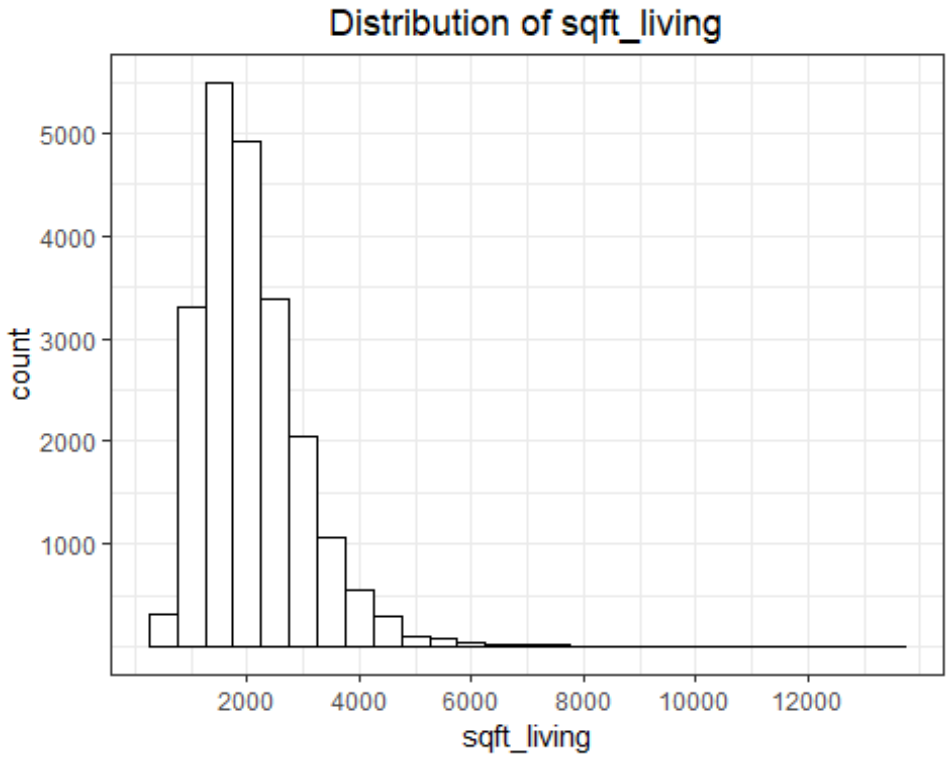
Next, look at the summary statistics and distribution of sqft_living. We can see that the median house size is 1910 square feet, and the mean is 2080 square feet. The largest house is over 13,000 square feet.

```
summary(king$sqft_living)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     290    1427    1910    2080    2550   13540
```

The histogram shows that the distribution is positively skewed.

```
ggplot(king, aes(x=sqft_living)) +
  geom_histogram(binwidth=500,  color="black", fill="white") +
  scale_x_continuous(breaks=c(2000, 4000, 6000, 8000, 10000, 12000)) +
  scale_y_continuous(breaks=c(1000, 2000, 3000, 4000, 5000)) +
  labs(title="Distribution of sqft_living") +
  theme_bw() +
  theme(plot.title=element_text(hjust=0.5))
```



d. bedrooms (the number of bedrooms in each house)

Use the table() function to obtain counts of the number of bedrooms.

```
table(king$bedrooms)

##
##    0    1    2    3    4    5    6    7    8    9   10   11   33
##   13  199 2760 9824 6882 1601  272   38   13    6    3    1    1
```

In the table, there is one house that has 33 bedrooms. This is probably an input error and was meant to be 3 bedrooms, so let's change it to 3 bedrooms:

```
which(king$bedrooms==33)

## [1] 15871

king$bedrooms[15871] = 3
table(king$bedrooms)

##
##    0    1    2    3    4    5    6    7    8    9   10   11
##   13  199 2760 9825 6882 1601  272   38   13    6    3    1
```

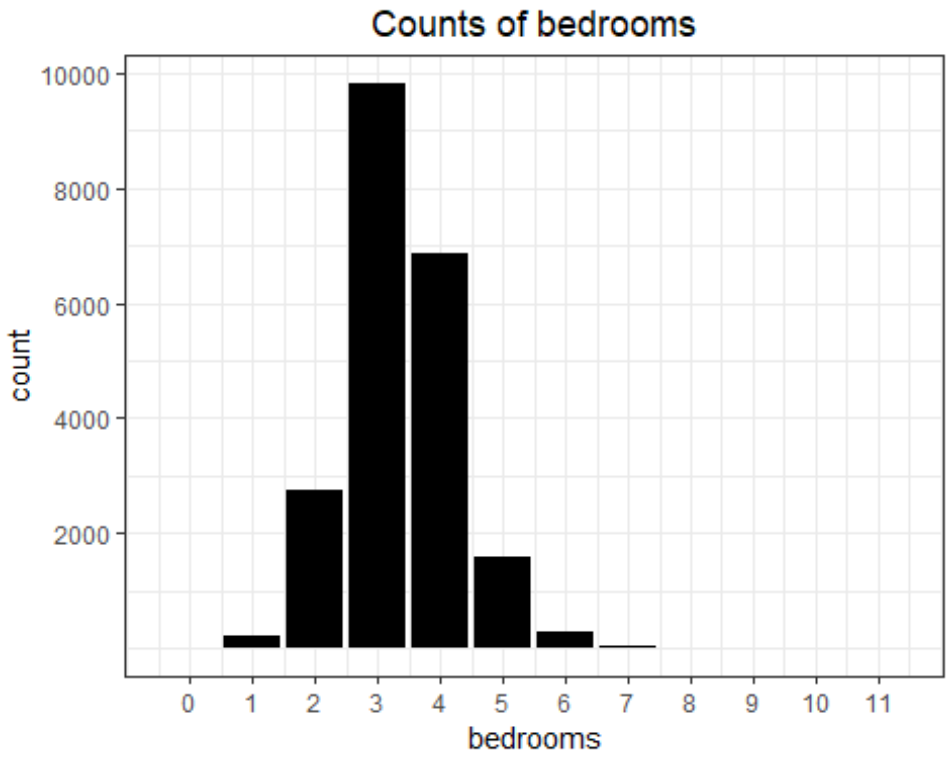
Now examine the summary statistics and the distribution of bedrooms. From the summary, the median number of bedrooms is 3, and the mean is 3.369.

```
summary(king$bedrooms)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   3.000   3.000   3.369   4.000  11.000
```

We use geom_bar() in ggplot to plot the distribution of bedrooms.

```
ggplot(king, aes(x=bedrooms))+
  geom_bar(fill="black") +
  scale_x_continuous(breaks=c(0,1,2,3,4,5,6,7,8,9,10,11)) +
  scale_y_continuous(breaks=c(2000, 4000, 6000, 8000, 10000)) +
  labs(title="Counts of bedrooms") +
  theme_bw() +
  theme(plot.title=element_text(hjust=0.5))
```



e. bathrooms (the number of bathrooms in each house)

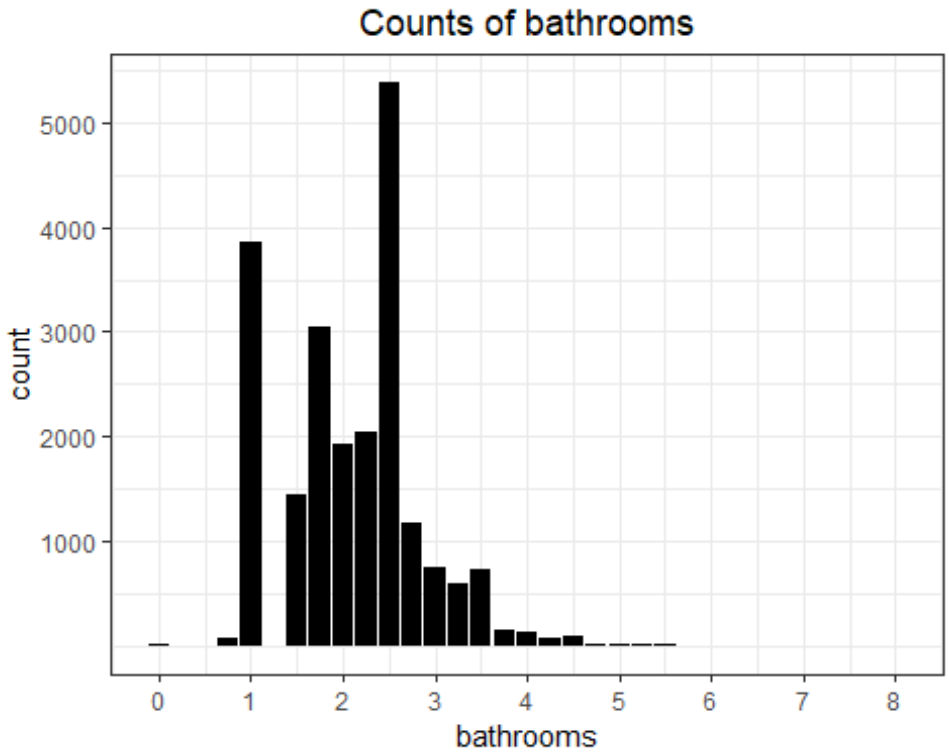
Next, examine the summary statistics of bathrooms. The median number of bathrooms is 2.250, and the mean is 2.115.

```
summary(king$bathrooms)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   1.750   2.250   2.115   2.500   8.000
```

Plot the distribution of bathrooms with geom_bar().

```
ggplot(king, aes(x=bathrooms)) +
  geom_bar(fill="black") +
  scale_x_continuous(breaks=c(0,1,2,3,4,5,6,7,8)) +
  scale_y_continuous(breaks=c(1000, 2000, 3000, 4000, 5000)) +
  labs(title="Counts of bathrooms") +
  theme_bw() +
  theme(plot.title=element_text(hjust=0.5))
```



f. grade (quality of the house)

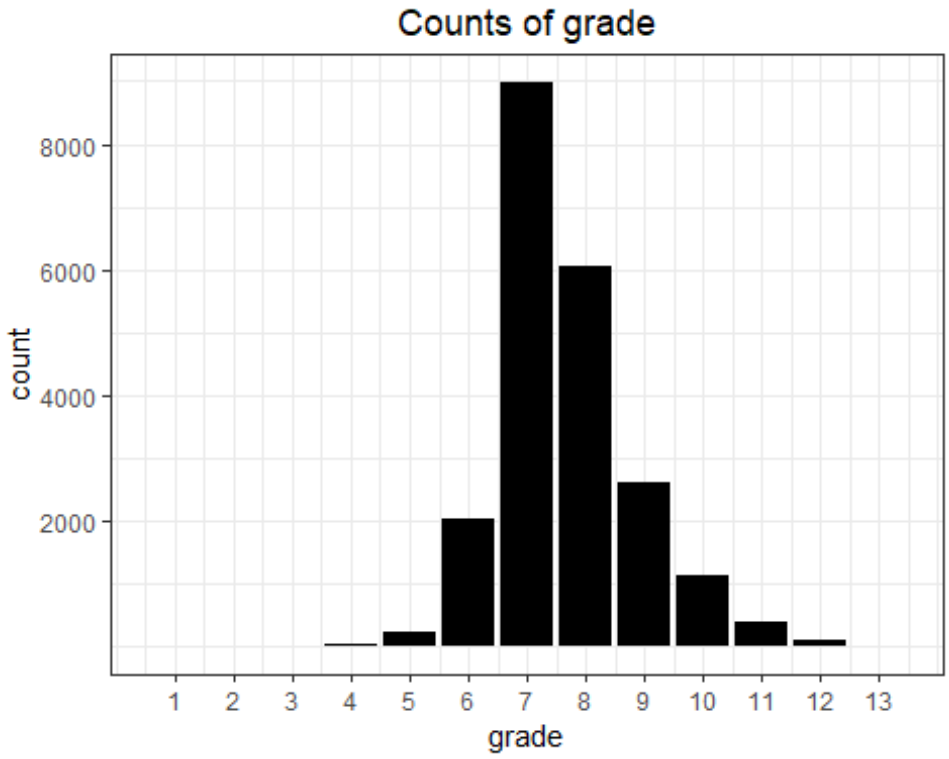
The grade of a house is based on the King County grading system, and it is given based on the quality of the house. Higher values indicate a nicer house. The median grade is 7, and the mean is 7.657.

```
summary(king$grade)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.000   7.000   7.000   7.657   8.000  13.000
```

Plot the distribution of grade with geom_bar().

```
ggplot(king, aes(x=grade)) +
  geom_bar(fill="black") +
  scale_x_continuous(breaks=c(1,2,3,4,5,6,7,8,9,10,11,12,13)) +
  scale_y_continuous(breaks=c(2000, 4000, 6000, 8000, 10000)) +
  labs(title="Counts of grade") +
  theme_bw() +
  theme(plot.title=element_text(hjust=0.5))
```



---

4. Examine the relationships among the variables

Before modeling the data, we need to get a sense of how the house features are related to price. Since we will model the prediction of sqrt_price rather than price, we will plot the relationship between sqrt_price and a few of the the predictors below.

a. correlation plot

First, we'll plot the correlations among the variables using the corrplot() function from the corrplot library.

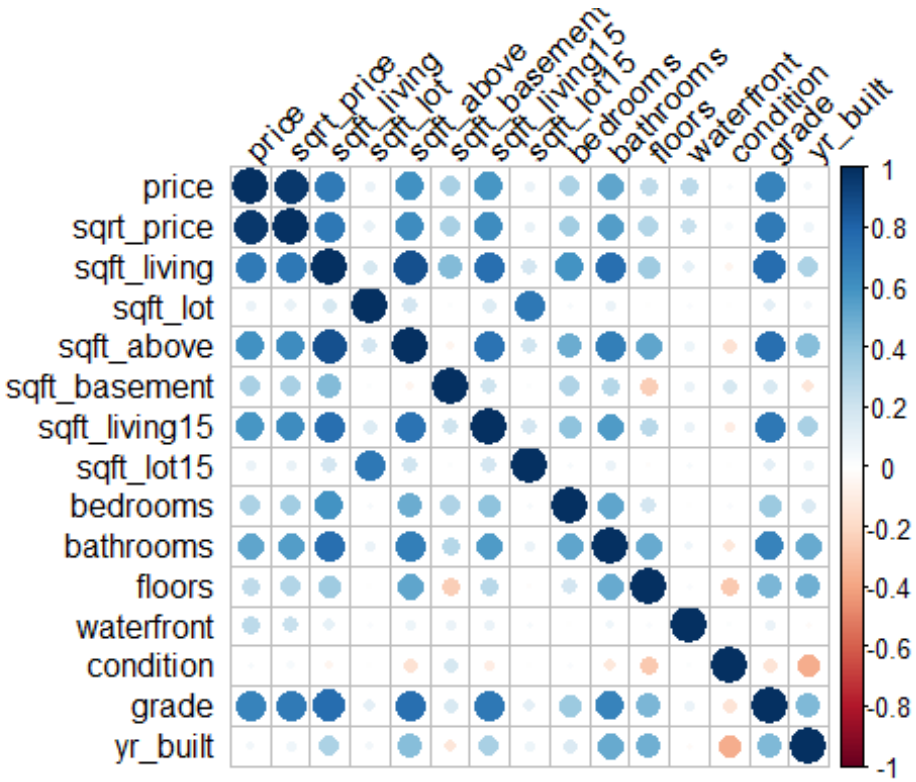
```
library(corrplot)
```

First create a separate data frame containing only the variables of interest. Then compute their correlations.

```
king.cor <- with(king, cbind(price, sqrt_price, sqft_living, sqft_lot, sqft_above,
                             sqft_basement, sqft_living15, sqft_lot15, bedrooms, bathroom
s,
                             floors, waterfront, condition, grade, yr_built))
cors <- cor(king.cor)
round(cors, digits=2)
```

Now plot the correlations. We can see in the correlation plot below that many of the features are positively correlated with each other. In particular, sqft_living, sqft_living15, bathrooms, and grade are highly positively correlated with price and sqrt_price.

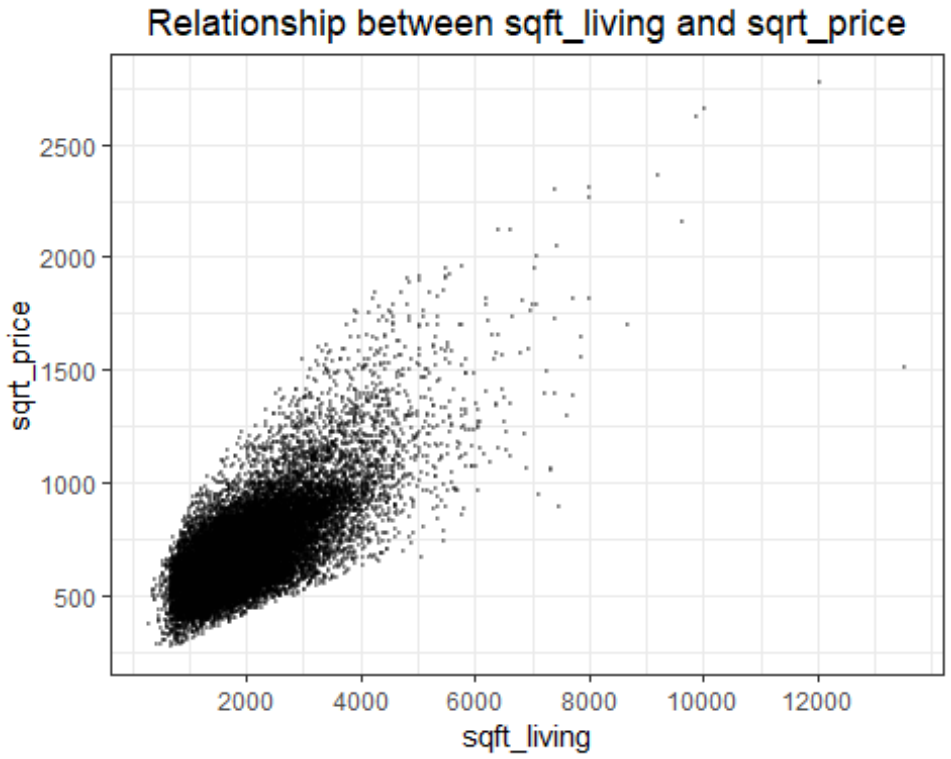
```
corrplot(cors, tl.col = "black", tl.srt = 45)
```



b. relationship between sqft_living and sqrt_price

Here, we create a scatter plot using geom_point() to visualize the relationship between sqft_living and sqrt_price. You can see that there is a positive relationship between square footage and house price; the bigger the house, the more expensive it is.

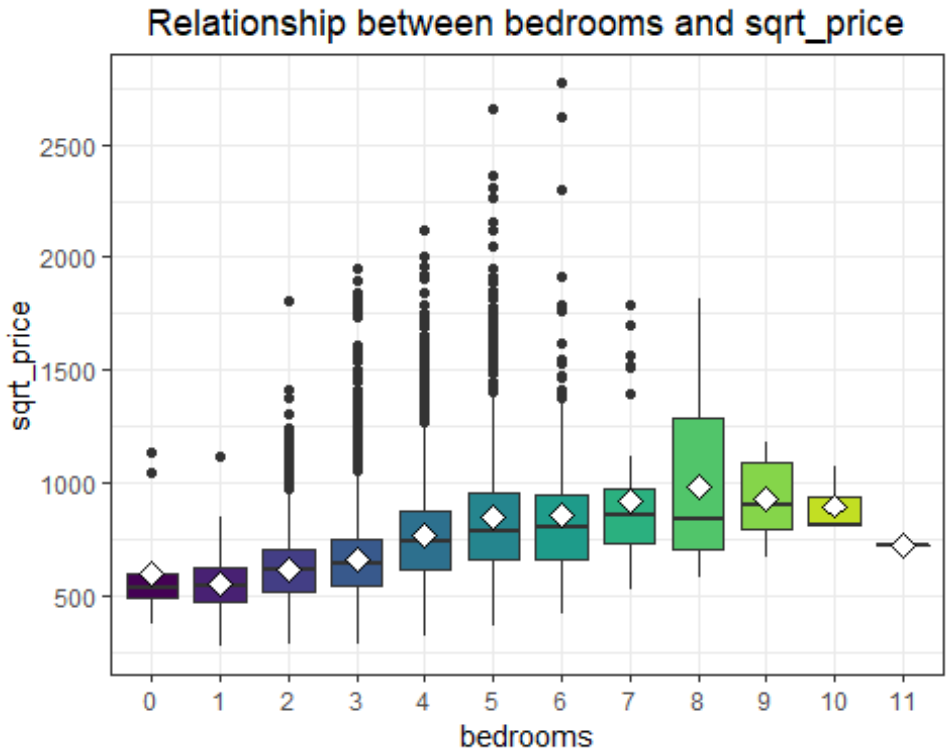
```
ggplot(king, aes(x=sqft_living, y=sqrt_price)) +
  geom_point(size=.5, alpha=.3) +
  scale_x_continuous(breaks=c(2000, 4000, 6000, 8000, 10000, 12000)) +
  scale_y_continuous(breaks=c(500, 1000, 1500, 2000, 2500)) +
  labs(title="Relationship between sqft_living and sqrt_price") +
  theme_bw() +
  theme(plot.title=element_text(hjust=0.5))
```



c. relationship between bedrooms and sqrt_price

Next, we create a box plot to illustrate relationship between bedrooms and sqrt_price using geom_boxplot(). The price appears to increase with more bedrooms.

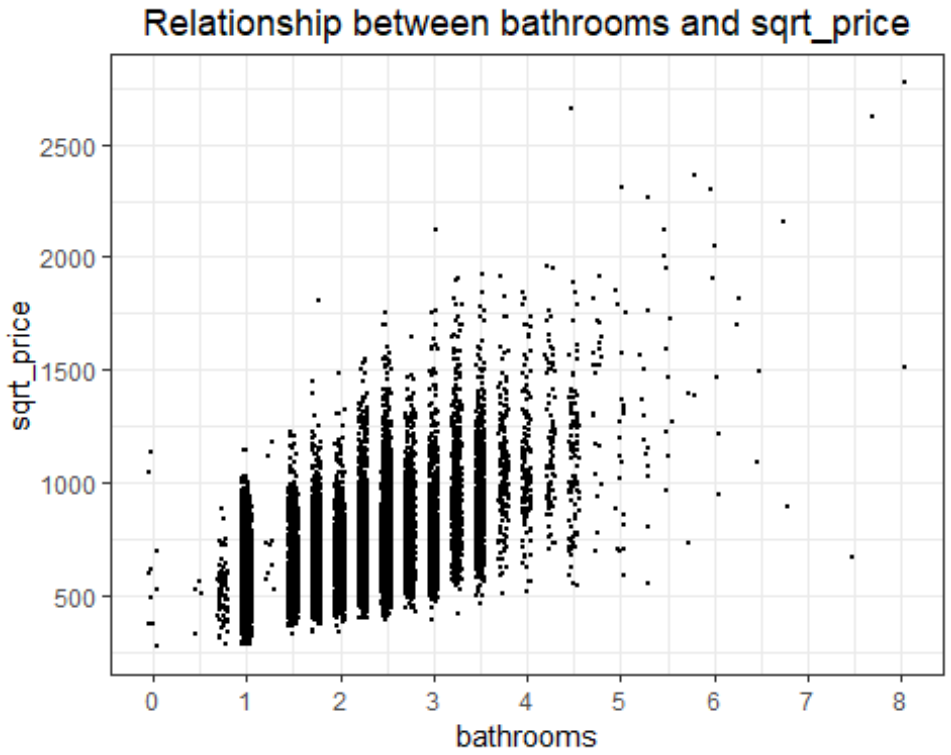
```
ggplot(king, aes(x=factor(bedrooms), y=sqrt_price)) +
  geom_boxplot(aes(fill=factor(bedrooms))) +
  stat_summary(fun.y="mean", geom="point", shape=23, size=3, fill="white") +
  scale_y_continuous(breaks=c(500, 1000, 1500, 2000, 2500)) +
  scale_fill_viridis_d() +
  guides(fill=FALSE) +
  labs(x="bedrooms", title="Relationship between bedrooms and sqrt_price") +
  theme_bw() +
  theme(plot.title=element_text(hjust=0.5))
```



d. relationship between bathrooms and sqrt_price

A scatterplot of bathrooms and sqrt_price shows that house prices also increase with more of bathrooms.

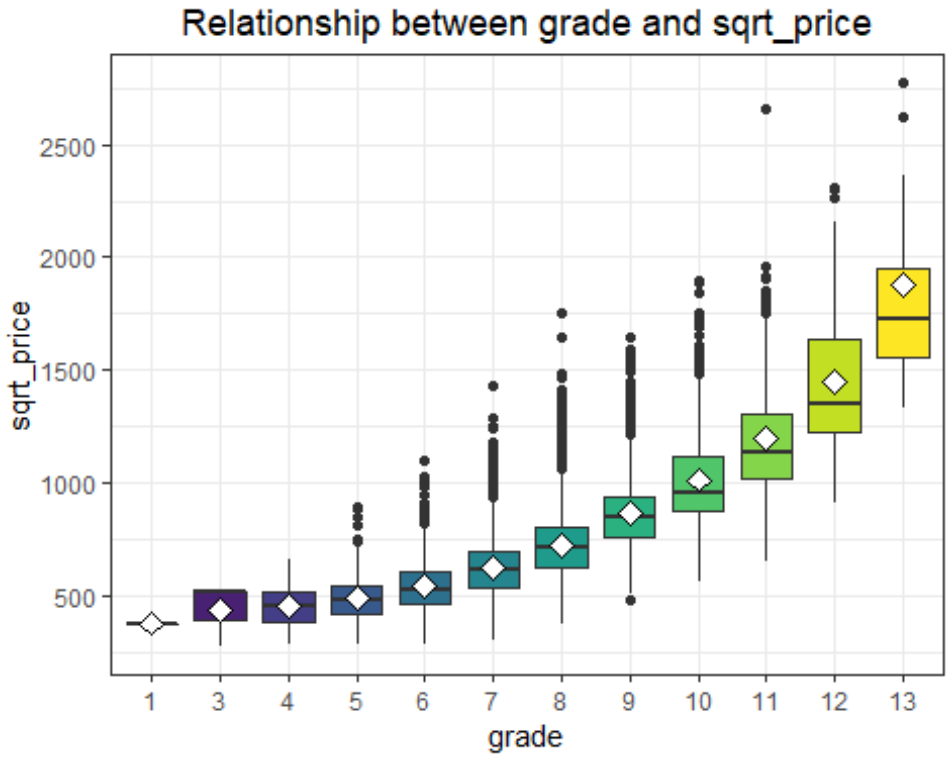
```
ggplot(king, aes(x=bathrooms, y=sqrt_price)) +
  geom_point(size=.5, position=position_jitter(width=.05)) +
  scale_x_continuous(breaks=c(0,1,2,3,4,5,6,7,8)) +
  scale_y_continuous(breaks=c(500, 1000, 1500, 2000, 2500)) +
  labs(title="Relationship between bathrooms and sqrt_price") +
  theme_bw() +
  theme(plot.title=element_text(hjust=0.5))
```



e. relationship between grade and sqrt_price

Finally, a box plot of grade and sqrt_price indicates clear positive relationship between grade and sqrt_price:

```
ggplot(king, aes(x=factor(grade), y=sqrt_price)) +
  geom_boxplot(aes(fill=factor(grade))) +
  stat_summary(fun.y="mean", geom="point", shape=23, size=3, fill="white") +
  scale_y_continuous(breaks=c(500, 1000, 1500, 2000, 2500)) +
  scale_fill_viridis_d() +
  guides(fill=FALSE) +
  labs(x="grade", title="Relationship between grade and sqrt_price") +
  theme_bw() +
  theme(plot.title=element_text(hjust=0.5))
```



---

5. Fit statistical models

Before fitting any models, we need to create a separate data frame containing only the variables to be included in the models. We will predict sqrt_price from sqft_living, sqft_lot, sqft_living15, sqft_lot15, bedrooms, bathrooms, floors, waterfront, condition, grade, and yr_built, so we will subset these variables into a new data frame called king2:

```
king2 <- king[,c("sqrt_price", "sqft_living", "sqft_lot", "sqft_living15", "sqft_lot15",
                 "bedrooms", "bathrooms",  "floors", "waterfront", "condition", "grade",
                 "yr_built")]
```

Since we will use cross-validation to select our models and calculate model fit, we need to separate the data into a training set and a test set. For each of the models, we will use the training set to fit the model, then we will calculate predictions on the test set and compute the mean square error (MSE) as an indicator of model fit.

Be sure to set the seed before splitting the data so that you can replicate your results:

```
set.seed(9)
samp <- sample(nrow(king2), nrow(king2)/2)
train <- king2[samp,]
test <- king2[-samp,]
```

a. Best subset selection

Best subset selection is a method for selecting the best set of predictors in a linear regression model. For each possible number of predictors in the model, the combination of predictors that results in the best fitting model, as measured by the amount of variance explained in the outcome, is selected. For example, since we have p=11 predictors, best subset selection will select the best predictor for the model having

p=1 predictor, then the best combination of predictors for the model with p=2 predictors, and so on, up to the model with all 11 predictors included.

To perform subset selection, we need to use the regsubsets() function from the leaps library:

```
library(leaps)
```

Now fit the model on the training data.

```
best.fit2 <- regsubsets(sqrt_price ~ ., data=train, nvmax=11)
```

View a summary of the model. The summary shows the variables (marked with an asterisk *) that are associated with the best-fitting model of each size, quantified by residual sum of squares (RSS). For example, the best fitting model with three predictors includes sqft_living, grade, and yr_built.

```
best.summary2 <- summary(best.fit2)
best.summary2
```

```
## Subset selection object
## Call: regsubsets.formula(sqrt_price ~ ., data = train, nvmax = 11)
## 11 Variables  (and intercept)
##               Forced in Forced out
## sqft_living       FALSE      FALSE
## sqft_lot          FALSE      FALSE
## sqft_living15     FALSE      FALSE
## sqft_lot15        FALSE      FALSE
## bedrooms          FALSE      FALSE
## bathrooms         FALSE      FALSE
## floors            FALSE      FALSE
## waterfront        FALSE      FALSE
## condition         FALSE      FALSE
## grade             FALSE      FALSE
## yr_built          FALSE      FALSE
## 1 subsets of each size up to 11
## Selection Algorithm: exhaustive
##           sqft_living sqft_lot sqft_living15 sqft_lot15 bedrooms bathrooms
## 1  ( 1 )  "*"         " "      " "           " "         " "      " "
## 2  ( 1 )  "*"         " "      " "           " "         " "      " "
## 3  ( 1 )  "*"         " "      " "           " "         " "      " "
## 4  ( 1 )  "*"         " "      " "           " "         " "      " "
## 5  ( 1 )  "*"         " "      " "           " "         " "      "*"
## 6  ( 1 )  "*"         " "      " "           " "         "*"      "*"
## 7  ( 1 )  "*"         " "      "*"           " "         "*"      "*"
## 8  ( 1 )  "*"         " "      "*"           " "         "*"      "*"
## 9  ( 1 )  "*"         " "      "*"           " "         "*"      "*"
## 10  ( 1 ) "*"         " "      "*"           "*"         "*"      "*"
## 11  ( 1 ) "*"         "*"      "*"           "*"         "*"      "*"
##           floors waterfront condition grade yr_built
## 1  ( 1 )  " "    " "        " "       " "   " "
## 2  ( 1 )  " "    " "        " "       "*"   " "
## 3  ( 1 )  " "    " "        " "       "*"   "*"
## 4  ( 1 )  " "    "*"        " "       "*"   "*"
## 5  ( 1 )  " "    "*"        " "       "*"   "*"
## 6  ( 1 )  " "    "*"        " "       "*"   "*"
## 7  ( 1 )  " "    "*"        " "       "*"   "*"
## 8  ( 1 )  "*"    "*"        " "       "*"   "*"
## 9  ( 1 )  "*"    "*"        "*"       "*"   "*"
```

```
## 10  ( 1 ) "*"    "*"        "*"       "*"   "*"
## 11  ( 1 ) "*"    "*"        "*"       "*"   "*"
```

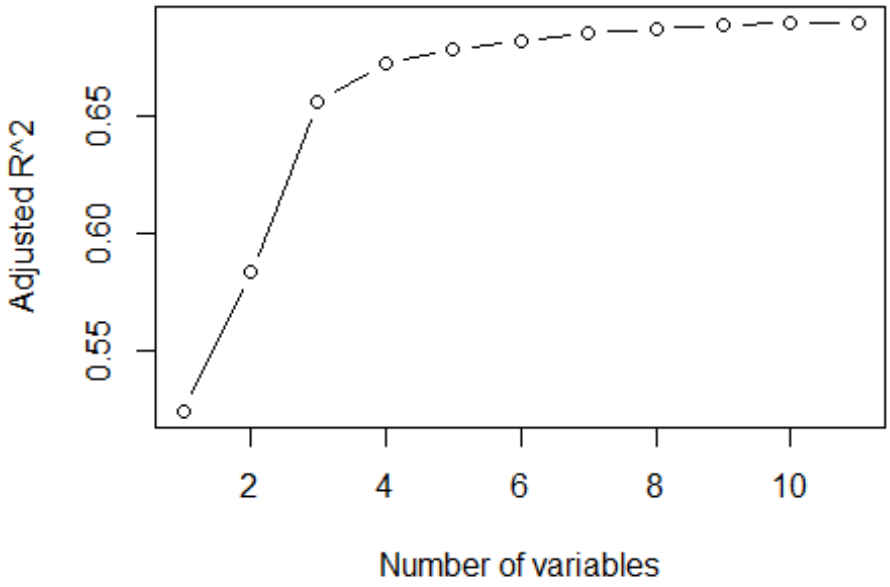
The model produces several fit statistics for the best fitting model of each size, including R-squared, adjusted R-squared, residual sum of squares, Mallows' Cp, and the Bayesian information criteria. We can look at adjusted R^2 for each model size by calling the adjr2 object. For example, for the model with three predictors, adjusted R^2 = 0.6561952.

```
best.summary2$adjr2
```

```
##  [1] 0.5238198 0.5829245 0.6561952 0.6722781 0.6779246 0.6820767 0.6851176
##  [8] 0.6869600 0.6884789 0.6894662 0.6895863
```

We can plot adjr2 to get an idea of how it changes across model sizes:

```
plot(best.summary2$adjr2, xlab="Number of variables", ylab="Adjusted R^2", type="b")
```



From the plot, it appears that the model fit increases sharply at 3 predictors, then begins to level off. However, the best fit based on adjusted R^2 occurs when all 11 predictors are included in the model:

```
which.max(best.summary2$adjr2)
```

```
## [1] 11
```

The adjusted R^2 value associated with 11 predictors is 0.6895863:

```
best.summary2$adjr2[11]
```

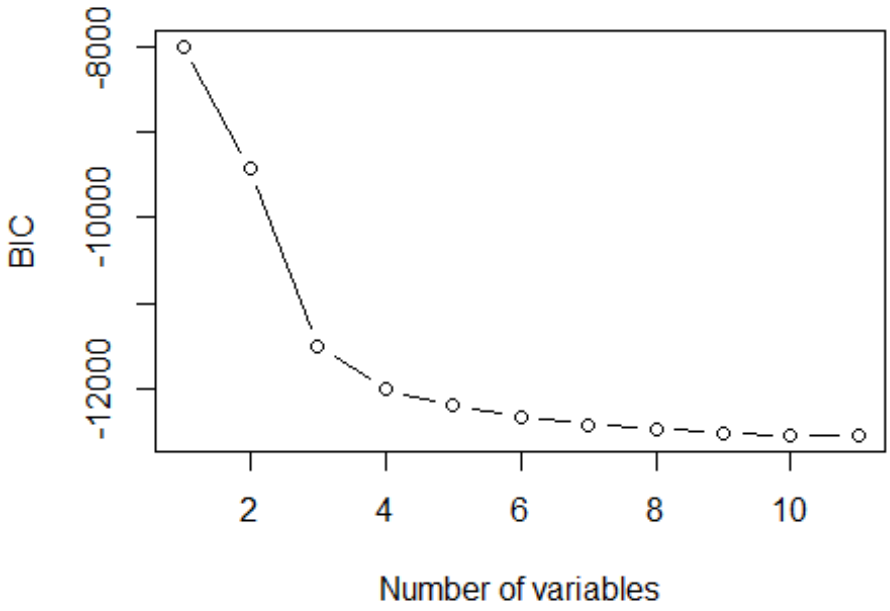
```
## [1] 0.6895863
```

We can also examine model fit by looking at the Bayesian Information Criterion (BIC).

```
best.summary2$bic
```

```
##  [1]  -8000.033  -9423.855 -11503.212 -12012.626 -12192.145 -12324.069
##  [7] -12419.637 -12474.765 -12519.036 -12545.050 -12540.944

plot(best.summary2$bic, xlab="Number of variables", ylab="BIC", type="b")
```



The BIC statistic also shows that the model fit improves dramatically around 3 or 4 predictors, then begins to level off. However, the best model based on BIC (BIC = -12545.05) includes only 10 predictors:

```
which.min(best.summary2$bic)

## [1] 10

best.summary2$bic[10]

## [1] -12545.05
```

Although the fit statistics show that a full model fits the data best, we need to fit the models on the test set to make sure we are not overfitting the model. First, we create a model matrix from the test set data. This model matrix will be used to compute the test set errors.

```
test.mat <- model.matrix(sqrt_price ~ ., data=test)
```

Now we fit the model on the test set matrix and compute the test set error for the best model of each model size. For each model size, we extract the coefficients from best.fit2, multiply them into the appropriate columns of the test model matrix to form the predictions, compute the test mean square error (MSE), then add the test MSE to a vector called val.errors.

```
val.errors <- rep(NA, 11)
for (i in 1:11){
  coefi <- coef(best.fit2, id=i)
  pred <- test.mat[,names(coefi)] %*% coefi
  val.errors[i] <- mean((test$sqft_living-pred)^2)
}
```

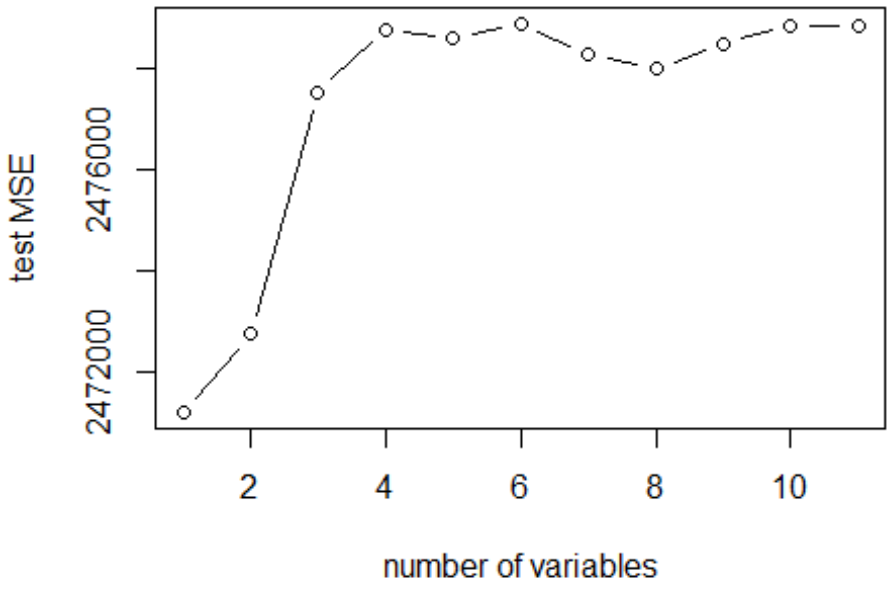
Now we can view the cross-validation errors for each model size:

```
val.errors
```

```
##  [1] 2471200 2472748 2477549 2478788 2478622 2478906 2478283 2478016
##  [9] 2478519 2478878 2478869
```

Plot the errors for each model size.

```
plot(val.errors, xlab="number of variables", ylab="test MSE", type="b")
```



As we can see, the model for which the test MSE is smallest has only one predictor:

```
which.min(val.errors)
```

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

The error associated with the 1-predictor model is 2,471,200:

```
val.errors[1]
```

```
## [1] 2471200
```

Based on the test set MSE, the model with only one predictor fits the best, in sharp contrast to what we found when we only looked at results on the training set. We can obtain the coefficients for this model using coef():

```
coef(best.fit2, 1)
```

```
## (Intercept) sqft_living
## 370.6955487   0.1611133
```

Finally, to obtain the most accurate model coefficients, perform a linear regression on the entire data set using the predictor obtained for the best 1-variable model.

```
best.lm <- lm(sqrt_price ~ sqft_living, data=king2)
summary(best.lm)
```

```
##
## Call:
## lm(formula = sqrt_price ~ sqft_living, data = king2)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1018.56   -99.79    -8.17    80.61   790.96
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 3.756e+02  2.376e+00   158.1   <2e-16 ***
## sqft_living 1.590e-01  1.045e-03   152.1   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 141.1 on 21611 degrees of freedom
## Multiple R-squared:  0.5172, Adjusted R-squared:  0.5172
## F-statistic: 2.315e+04 on 1 and 21611 DF,  p-value: < 2.2e-16
```

b. forward and backward stepwise regression

Like best subset selection, forward and backward stepwise regression select the best subset of predictors for each model size. However, it takes a "greedy" approach to selection. Forward stepwise selection selects the best model of each size by successively adding predictors in the model. It starts with a model with no predictors, then adds one predictor at a time by selecting the predictor that adds the greatest improvement to the model, until all predictors are included in the model. In contrast, backward stepwise selection begins with all predictors in the model, then it removes one predictor at a time, selecting the one to remove contributes least to model fit, until there is only one predictor remaining.

Like best subset selection, forward and backward stepwise regression can be conducted using cross-validation. In this example, however, I'll just fit the models on the full data set. First we will perform forward stepwise selection by setting method="forward" in the regsubsets() function:

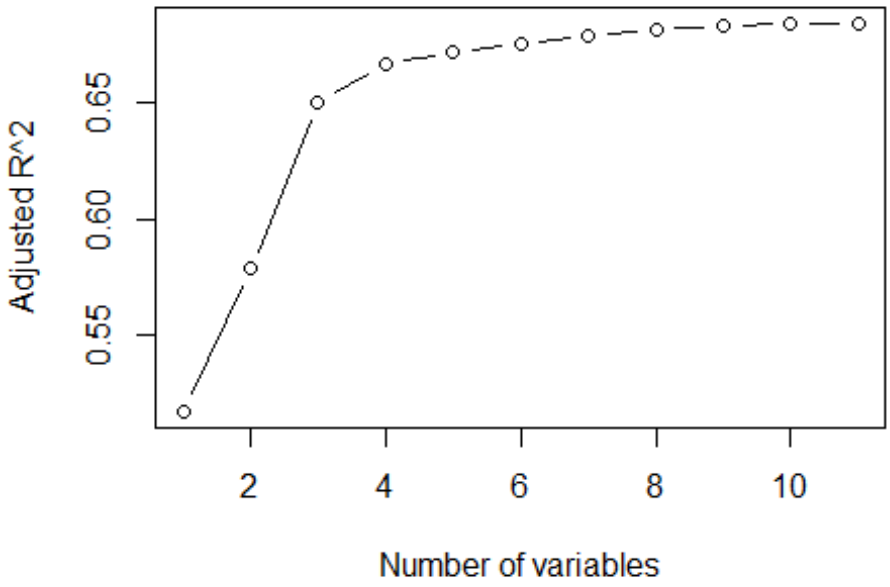
```
fwd.fit1 <- regsubsets(sqrt_price ~ ., data=king2, nvmax=11, method="forward")
```

Obtain adjusted R^2 for each model size, then plot these values.

```
fwd.summary1 <- summary(fwd.fit1)
fwd.summary1$adjr2
```

```
##  [1] 0.5171587 0.5784728 0.6502607 0.6666787 0.6717546 0.6756839 0.6791926
##  [8] 0.6815484 0.6833160 0.6841853 0.6842175
```

```
plot(fwd.summary1$adjr2, xlab="Number of variables", ylab="Adjusted R^2", type="b")
```



As with best subset selection, the fit increases sharply at 3 predictors, then begins to level off.

Finally, determine the model size for which the adjusted R^2 is greatest. Below, we see that the best fit based on adjr2 (0.6837) occurs when all 11 predictors are included in the model. (However, from what we saw when performing cross-validation with best subset selection, a model with all predictors included is likely overfitting the data.)

```
which.max(fwd.summary1$adjr2)

## [1] 11

fwd.summary1$adjr2[11]

## [1] 0.6842175
```

Next, We perform backward stepwise selection in the same way as forward stepwise selection, except that we specify method="backward":

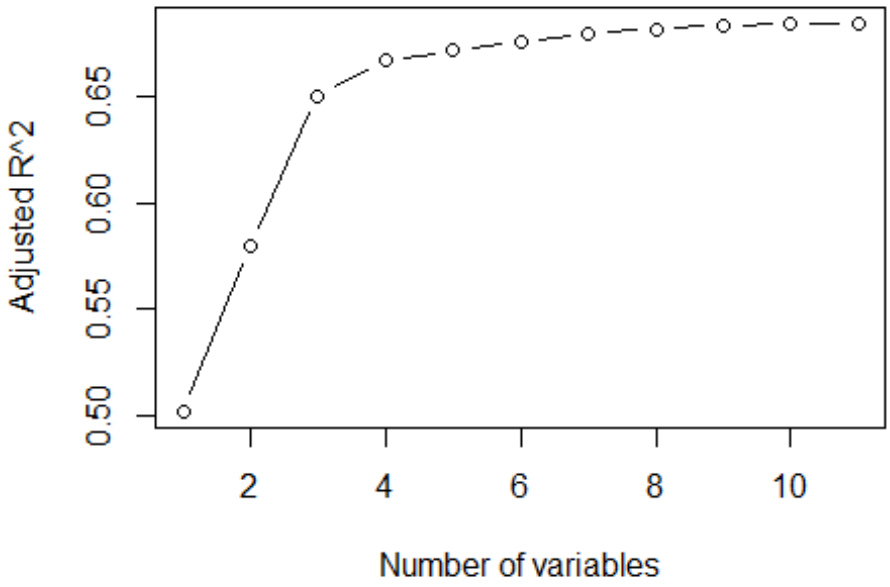
```
bwd.fit1 <- regsubsets(sqrt_price ~ ., data=king2, nvmax=11, method="backward")
```

Obtain adjusted R^2 for each model size, then plot these values.

```
bwd.summary1 <- summary(bwd.fit1)
bwd.summary1$adjr2

##  [1] 0.5021879 0.5798190 0.6502607 0.6666787 0.6717546 0.6756839 0.6791926
##  [8] 0.6815484 0.6833160 0.6841853 0.6842175

plot(bwd.summary1$adjr2, xlab="Number of variables", ylab="Adjusted R^2", type="b")
```



Again, the fit increases sharply at 3 predictors, then levels off. The best fit based on adjr2 occurs when all 11 predictors are included in the model:

```
which.max(bwd.summary1$adjr2)

## [1] 11

bwd.summary1$adjr2[11]

## [1] 0.6842175
```

c. ridge regression

Ridge regression and lasso are a shrinkage methods that shrink, or constrain, the size of regression coefficients by imposing a lambda penalty on the model. Although these methods impose some bias in the models, they reduce variance across models and prevent over-fitting.

Ridge regression shrinks the coefficients by imposing contraint with a distance measure called the L2 norm. This means that coefficients can be shrunken to be very close to zero, but they will rarely ever equal zero. Thus, ridge regression models will likely include all predictors in the model, but the coefficients of some of these predictors will be very small.

We use the the cv.glmnet() function from the glmnet library to run ridge regression with 10-fold cross-validation. Cross-validation is performed in order to select the best value of lambda.

```
library(glmnet)
```

cv.glmnet() takes an x matrix and a y vector. Create a model matrix for x and a vector for y for the training and test sets.

```
xtrain <- model.matrix(sqrt_price ~ ., data=train)
ytrain <- train$sqrt_price
xtest <- model.matrix(sqrt_price ~ ., data=test)
ytest <- test$sqrt_price
```

Since we are using cross-validation to select a value of lambda, create a grid of lambda values to feed into the model. Here, we create a vector of 100 values that range from 10^10 to 10^-2:

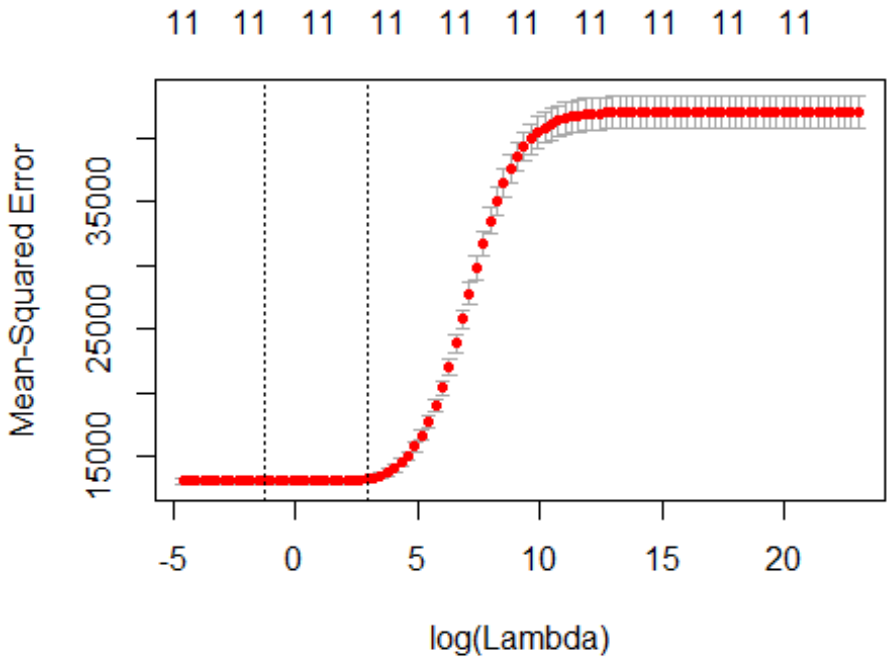
```
grid <- 10^seq(10, -2, length=100)
```

Now we fit ridge regression on the training set. The cv.glmnet() function performs 10-fold CV by default, and it automatically standardizes the variables before fitting the model. The alpha=0 argument tells glmnet to fit a ridge regression (as opposed to alpha=1, which tells glmnet to fit a lasso model).

```
set.seed(9)
ridge.cv3 <- cv.glmnet(xtrain, ytrain, alpha=0, lambda=grid)
```

The model calculates the cross-validated mean square error obtained for each value of lambda. We can plot these values using the plot() function. (Note that the MSE is plotted against the log of lambda rather than lambda itself.)

```
plot(ridge.cv3)
```



Identify the the minimum cross-validation error (cvm) and the lambda associated with that minimum value.

```
which.min(ridge.cv3$cvm)

## [1] 88

ridge.cv3$cvm[88]

## [1] 13097.58

ridge.cv3$lambda.min

## [1] 0.2848036
```

The lambda value that resulted in the lowest error was λ = 0.2848036, and the error associated with this value was 13097.58. Now we need to see how this compares to performance on the test set. On the test set, calculate predictions for each observation based on the model with the chosen value of lambda, then calculate the mean square error (MSE).

```
bestlam3 <- ridge.cv3$lambda.min
ridge.pred3 <- predict(ridge.cv3, s=bestlam3, newx=xtest)
mean((ytest-ridge.pred3)^2)

## [1] 13005.98
```

The minimum cross-validated MSE from the training set was 13097.58, and the MSE on the test set was 13005.98.

Finally, in order to obtain the most accurate coefficients for the model, we refit the ridge regression on the full data set using the glmnet() function. We set the lambda parameter to be equal to the best value of lambda chosen by cross-validation. (Note that the a0 object calls the intercept of the model, and the beta object calls the coefficients for the predictors.)

```
x <- model.matrix(sqrt_price ~ ., data=king2)
y <- king2$sqrt_price
ridge.fit3 <- glmnet(x, y, alpha=0, lambda=bestlam3)
ridge.fit3$a0

##       s0
## 4321.494

ridge.fit3$beta

## 12 x 1 sparse Matrix of class "dgCMatrix"
##                            s0
## (Intercept)    .
## sqft_living    7.254571e-02
## sqft_lot       4.825702e-05
## sqft_living15  3.237017e-02
## sqft_lot15    -2.793335e-04
## bedrooms      -1.846551e+01
## bathrooms      3.330384e+01
## floors         2.450315e+01
## waterfront     2.872624e+02
## condition      1.452258e+01
## grade          7.674568e+01
## yr_built      -2.289031e+00
```

d. lasso

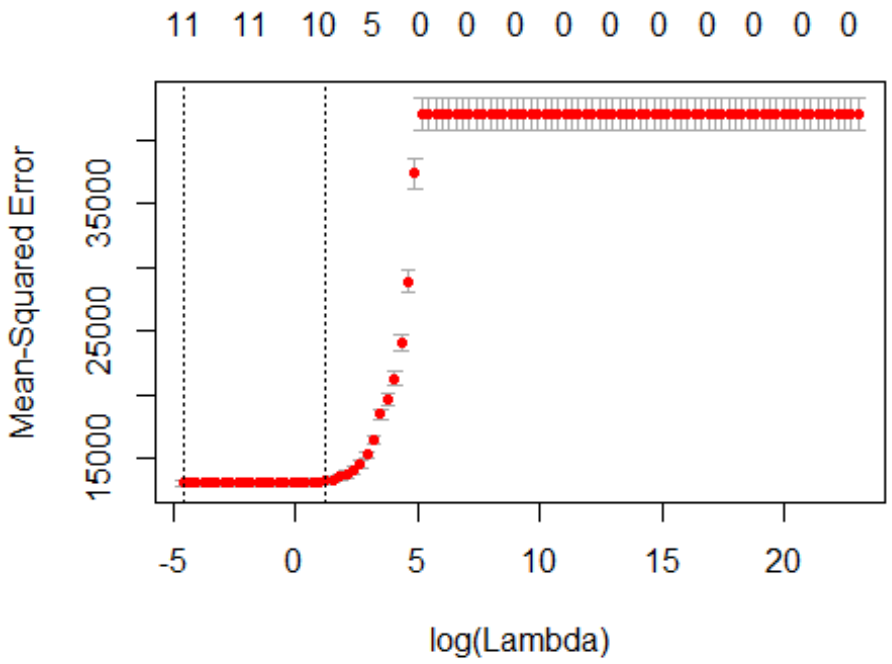
Lasso, like ridge regression, is a method that shrinks the size of regression coefficients by imposing a lambda penalty on the model. However, whereas ridge regression uses the L2 norm to constrain the size of coefficients, lasso uses the L1 norm. Thus, with lasso, the lambda penalty can shrink coefficients to zero, thereby removing those parameters from the model. Like ridge regression, lasso adds bias to the model but reduces variance by preventing over-fitting of the data.

As with ridge regression, we use the cv.glmnet() function from the glmnet library to perform lasso with cross-validation and to select the best value of lambda for the model. We'll first fit the model on the training set using the same xtrain matrix and ytrain vector and the same grid of lambda values that we had created for ridge regression:

```
set.seed(9)
lasso.cv3 <- cv.glmnet(xtrain, ytrain, alpha=1, lambda=grid)
```

Plot the cross-validated mean square error obtained for each value of lambda.

```
plot(lasso.cv3)
```



Identify the the minimum cross-validation error and the lambda associated with that minimum value.

```
which.min(lasso.cv3$cvm)

## [1] 100

lasso.cv3$cvm[100]

## [1] 13097.67

lasso.cv3$lambda.min

## [1] 0.01
```

The best value of lambda (λ=.01) is the smallest value from our grid of lambda values, with a corresponding cv error of 13097.67. We could consider creating a new grid that expands the range of values. However, as we can see in the plot, it appears that we are reaching an asymptote. The model seems to favor a small lambda penalty (and hence, the inclusion of more predictors in the data set and larger coefficients for these predictors). For now, we'll stick with lambda=0.01.

On the test set, calculated predictions for each observation based on the model with the best lambda value, then compute the MSE.

```
bestlam3 <- lasso.cv3$lambda.min
lasso.pred3 <- predict(lasso.cv3, s=bestlam3, newx=xtest)
mean((ytest-lasso.pred3)^2)
```

```
## [1] 13008.01
```

The minimum cross-validated MSE from the training set was 13097.67, and the MSE from the test set was 13008.01. These values are similar to the error values obtained when we ran ridge regression.

Finally, in order to obtain the most accurate coefficients for the model, we refit the lasso on the full data set using the glmnet() function. We set the lambda parameter to be equal to the best value of lambda chosen by cross-validation.

```
lasso.fit3 <- glmnet(x, y, alpha=1, lambda=bestlam3)
lasso.fit3$a0

##       s0
## 4331.952

lasso.fit3$beta

## 12 x 1 sparse Matrix of class "dgCMatrix"
##                            s0
## (Intercept)    .
## sqft_living    7.268738e-02
## sqft_lot       4.747425e-05
## sqft_living15  3.214313e-02
## sqft_lot15    -2.788454e-04
## bedrooms      -1.855279e+01
## bathrooms      3.327918e+01
## floors         2.448539e+01
## waterfront     2.872797e+02
## condition      1.447334e+01
## grade          7.695627e+01
## yr_built      -2.294799e+00
```

e. principal components regression (PCR) and partial least squares (PLS)

Principal components regression (PCR) is dimension reduction method that fits linear regression on principal components, or linear combinations of the predictors. These linear combinations are identified to be those along which the data vary the most. This method is especially useful for reducing the complexity of high-dimensional data.

Partial least squared (PLS) is similar to PCR, except that it identifies the linear combinations of the predictors by taking into account their relationship with the outcome variable. It places a higher weight on predictors that are more strongly related to the outcome.

We use the pls library to run PCR and PLC:

```
library(pls)
```

First, we fit PCR on the training set using the pcr() function from the pls library. Set scale=TRUE to standardize each predictor, and validation="CV" to compute 10-fold CV for each possible value of M, the number of principal components used in the regression.

```
set.seed(9)
pcr.fit2 <- pcr(sqrt_price ~ ., data=train, scale=TRUE, validation="CV")
```

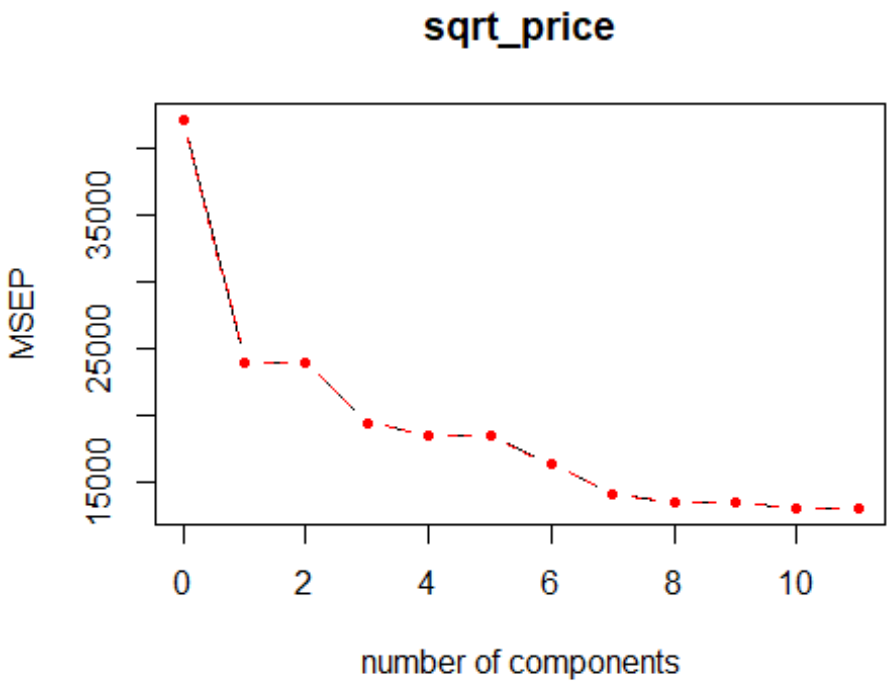
Now view the summary of the model. The summary provides two pieces of information for each number of components in the model: 1) the CV error in terms of root mean square error (RMSE) and 2) the % variance explained in the predictors and in the outcome variables.

```
summary(pcr.fit2)
```

```
## Data:    X dimension: 10806 11
##  Y dimension: 10806 1
## Fit method: svdpc
## Number of components considered: 11
##
## VALIDATION: RMSEP
## Cross-validated using 10 random segments.
##        (Intercept)  1 comps  2 comps  3 comps  4 comps  5 comps  6 comps
## CV           205.1      155    154.9    139.6    136.2    136.2    127.9
## adjCV        205.1      155    154.9    139.6    136.2    136.2    127.9
##        7 comps  8 comps  9 comps  10 comps  11 comps
## CV       119.1    116.1    116.1     114.5     114.5
## adjCV    119.1    116.1    116.1     114.5     114.5
##
## TRAINING: % variance explained
##             1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps
## X             37.40    52.88    65.24    74.33    80.75    86.68    91.10
## sqrt_price    42.94    43.06    53.71    56.01    56.02    61.24    66.37
##             8 comps  9 comps  10 comps  11 comps
## X             93.97    96.53     98.70    100.00
## sqrt_price    68.05    68.06     68.97     68.99
```

As we can see above, the lowest CV error occurs with 10 or 11 components (RMSE = 114.5). Plot the CV errors for each number of components in the regression using the validationplot() function. Set val.type="MSEP" to plot MSE instead of RMSE.

```
validationplot(pcr.fit2, val.type="MSEP", type="b", pch=20)
```



Finally, make predictions on the test set using the 10-component model that was fit on the training set. Then compute the mean square error (MSE) on the test set.

```
pcr.pred2 <- predict(pcr.fit2, test, ncomp=10)
mean( (test$sqrt_price-pcr.pred2)^2 )

## [1] 13015.64
```

We obtain a test MSE of 13015.64, similar to the error obtained from the models we've previously run.

Lastly, fit the PCR model on the full data set to examine the % variance explained when 10 components are included in the model. As we can see from the summary below, the 10-component model explains 98.71% of the variance in the predictors and 68.42% of the variance in the outcome:

```
pcr.fit3 <- pcr(sqrt_price ~ ., data=king2, scale=TRUE)
summary(pcr.fit3)

## Data:    X dimension: 21613 11
##  Y dimension: 21613 1
## Fit method: svdpc
## Number of components considered: 11
## TRAINING: % variance explained
##             1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps
## X             37.29    52.89    65.25    74.37    80.74    86.65    91.13
## sqrt_price    42.89    43.09    53.24    55.88    55.88    60.76    66.01
##             8 comps  9 comps  10 comps  11 comps
## X             94.00    96.52     98.71    100.00
## sqrt_price    67.59    67.59     68.42     68.44
```

To fit a partial least squares (PLS) model rather than PCR, we use the plsr() function from the pls library. We first fit the model on the training set. Set scale=TRUE to standardize each predictor, and validation="CV" to compute 10-fold CV for each possible value of M, the number of principal components used in the regression.

```
set.seed(9)
pls.fit1 <- plsr(sqrt_price ~ ., data=train, scale=TRUE, validation="CV")
```

View the model summary.

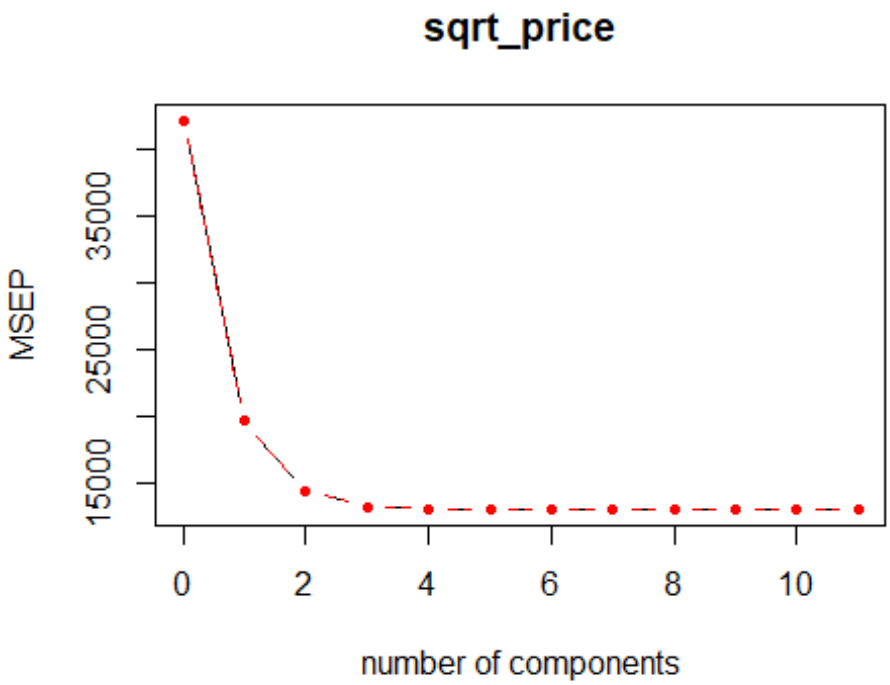
```
summary(pls.fit1)

## Data:    X dimension: 10806 11
##  Y dimension: 10806 1
## Fit method: kernelpls
## Number of components considered: 11
##
## VALIDATION: RMSEP
## Cross-validated using 10 random segments.
##        (Intercept)  1 comps  2 comps  3 comps  4 comps  5 comps  6 comps
## CV           205.1    140.3    120.3    115.3    114.7    114.5    114.5
## adjCV        205.1    140.3    120.3    115.3    114.7    114.5    114.5
##        7 comps  8 comps  9 comps  10 comps  11 comps
## CV       114.5    114.5    114.5     114.5     114.5
## adjCV    114.5    114.5    114.5     114.5     114.5
##
## TRAINING: % variance explained
##             1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps
## X             36.49    47.94    54.99    63.46    73.71    81.82    86.73
## sqrt_price    53.29    65.70    68.53    68.86    68.95    68.98    68.99
##             8 comps  9 comps  10 comps  11 comps
```

```
## X            89.41    91.20     97.43    100.00
## sqrt_price   68.99    68.99     68.99     68.99
```

The lowest cross-validation error occurs with 5-11 components (RMSE = 114.5). Use validationplot() to plot the errors against the number of components in the model:

```
validationplot(pls.fit1, val.type="MSEP", type="b", pch=20)
```



Since all the models that had between 5 to 11 components had the same RMSE, we'll use the 5-component model to make predictions on the test set and to compute the MSE on the test set.

```
pls.pred1 <- predict(pls.fit1, test, ncomp=5)
mean( (test$sqrt_price-pls.pred1)^2 )

## [1] 13020.61
```

We obtain a test MSE of 13020.61, similar to the PCR model. If we used a model with a larger number of components, we might obtain an even lower test MSE. However, this would come at the expense of having a more complex model.

Lastly, fit the PLS on the full data set to examine the % variance explained when 5 componenets are included in the model. (As we can see from the summary below, the 5-component model explains 74.57% of the variance in the predictors and 68.40% of the variance in the outcome.)

```
pls.fit2 <- plsr(sqrt_price ~ ., data=king2, scale=TRUE)
summary(pls.fit2)

## Data:    X dimension: 21613 11
##  Y dimension: 21613 1
## Fit method: kernelpls
## Number of components considered: 11
## TRAINING: % variance explained
##             1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps
## X             36.39    47.81    55.06     64.7    74.57    82.00    86.86
```

```
## sqrt_price    53.09    65.28    67.99     68.3    68.40    68.43    68.44
##             8 comps  9 comps  10 comps  11 comps
## X             89.53    91.11     97.47    100.00
## sqrt_price    68.44    68.44     68.44     68.44
```

f. random forest

Random forest is a decision tree method that makes predictions on an outcome by successively splitting the predictor space into separate regions, with each region having a different prediction on the outcome. Random forest builds many trees on bootstrapped samples of the data, then averages the predictions over all of the trees. For each split in a tree, only a random subset of predictors are considered for the split. This has the effect of decorrelating the trees and improving overall predictions. Random forests tend to have very good predictive accuracy.

We use the randomForest library to fit random forest models.

```
library(randomForest)
```

Fit a random forest model on the training data using the randomForest() function. By default, function randomly selects p/3 variables for each split in the regression tree.

```
set.seed(9)
rf.fit1 <- randomForest(sqrt_price ~ ., data=train, importance=TRUE)
```

Call the function name to review a summary of the model.

```
rf.fit1
```

```
##
## Call:
##  randomForest(formula = sqrt_price ~ ., data = train, importance = TRUE)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 3
##
##           Mean of squared residuals: 10521.96
##                     % Var explained: 74.99
```

The random forest model resulted in an MSE error of 10521.96. (Note that this error is calculated using out-of-bag estimates.)

View the importance statistics for each variable using the importance() function. The %IncMSE column indicates the mean decrease in accuracy of predictions on the out-of-bag samples when a given variable is excluded from the model. The IncNodePurity column is a measure of the total decrease in node impurity that results from splits over that variable, averaged over all trees.

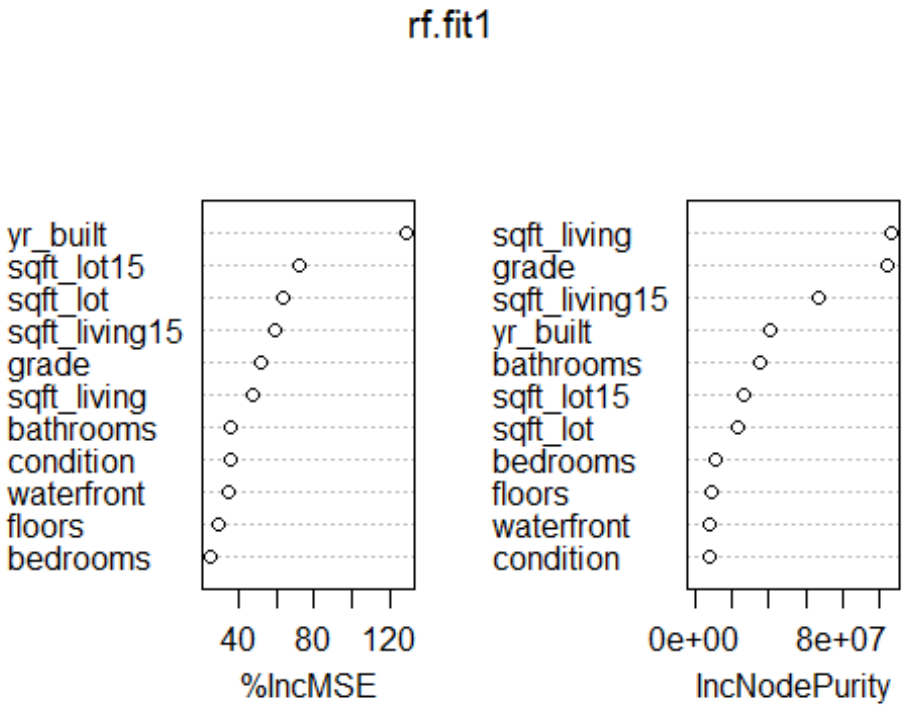
```
importance(rf.fit1)
```

```
##                 %IncMSE IncNodePurity
## sqft_living    47.32255     107228625
## sqft_lot       63.65156      23340156
## sqft_living15  59.76731      66609934
## sqft_lot15     72.36103      26717218
## bedrooms       25.82191      10941362
## bathrooms      35.72916      35401417
## floors         29.61705       8362110
## waterfront     34.85075       7975033
```

```
## condition      35.69176       7183765
## grade          52.23111     104278378
## yr_built      128.51652      40080494
```

We can plot the importance statistics using varImpPlot(). From the plots below, it appears that yr_built, sqft_lot, sqft_lot15, sqft_living, sqft_living15, and grade have the most influence on the model:

```
varImpPlot(rf.fit1)
```



Finally, determine how well the model performs on the test set by calculating predictions for the test set observations, then calculating the test set MSE.

```
rf.pred1 <- predict(rf.fit1, test)
mean( (test$sqrt_price-rf.pred1)^2 )

## [1] 10079.78
```

The random forest model resulted in a test MSE of 10079.78, much lower than the test set errors obtained from subset selection, ridge regression, lasso, PCR, and PLS.

---

6. Compare models

In this report, I showed how to perform best subset selection, forward and backward stepwise regression, ridge regression, lasso, principal components regression, partial least squares, and random forest in order to predict housing prices from various features of the house. In the table below, we can compare the performance of each of these models based on test MSE:

Method	Test MSE
Best subset	2,471,200
Ridge Regression	13,006

Lasso	13,008
PCR	13,016
PLS	13,021
Random forest	10,080

The random forest model performed the best, with the lowest test error of 10,080. Ridge regression, lasso, PCR, and PLS all performed similarly, with test errors around 13,000. Best subset selection performed the worst.