King County Regression

Eric Knudson

1/6/2020

```
library(dplyr)
library(ggplot2)
library(ggmap)
library(glmnet)
library(gridExtra)
library("leaps")
library(stringr)
```

This script precomputes the resources necessary to run the web application. It takes as input the raw data from Kaggle's King County Home Price Dataset and outputs:

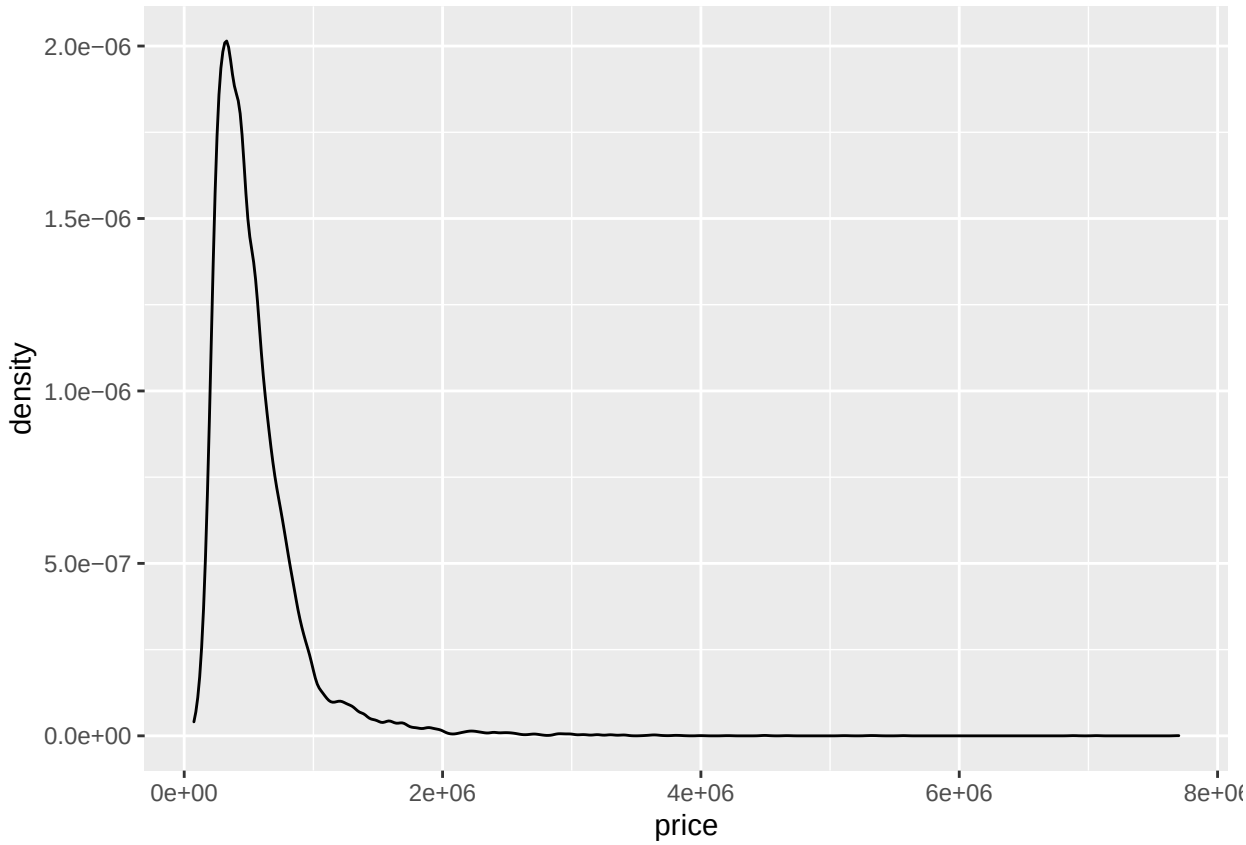
1. a cleaned version of that dataset, which is used by the web app when displaying nearby homes, similar homes, and in computing the histogram of home prices.
2. a linear model, which is used by the web app to predict the price of a user-input home.

Both of these are saved as `.rds` files in `/app/resources/`.

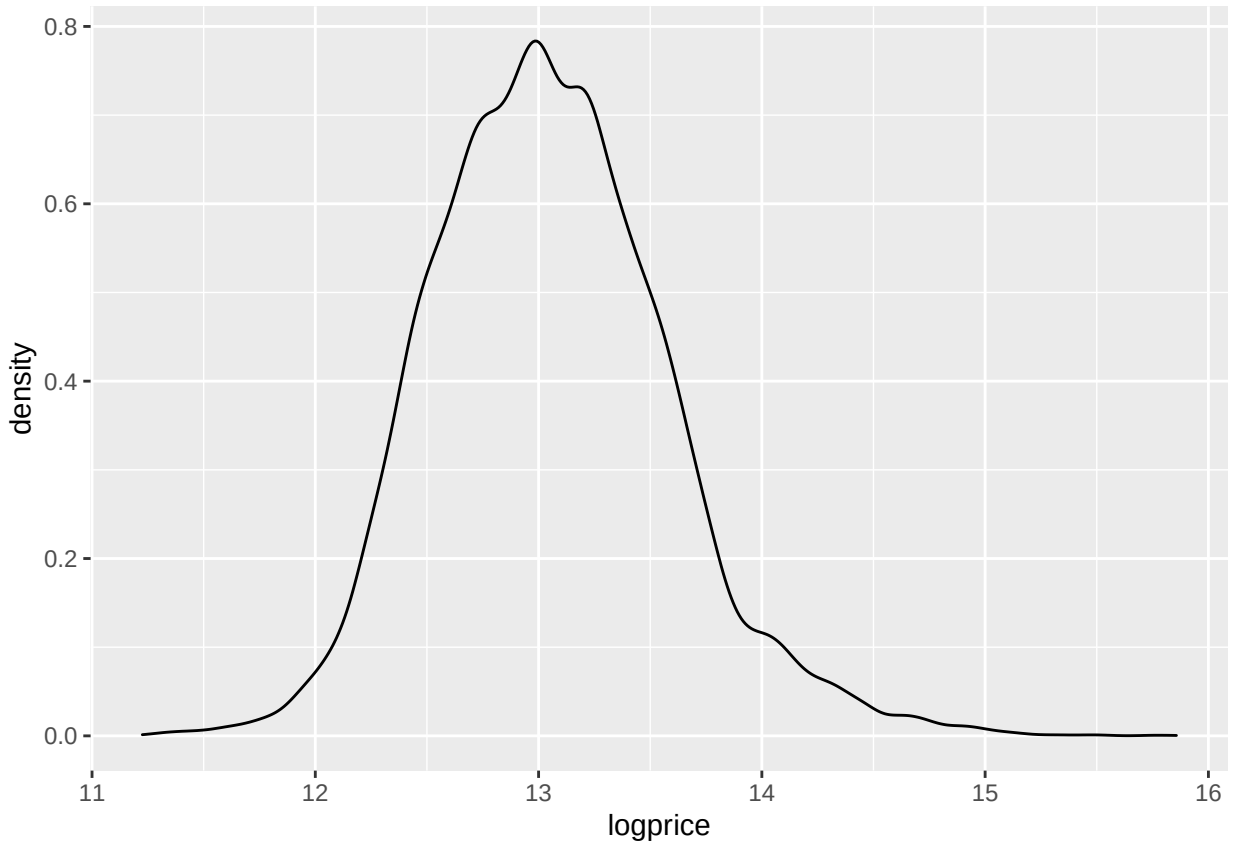
Data Import and Cleaning

Home prices have a highly skewed distribution, so I may want to use a log-linear model. A linear-linear model is also a theoretically plausible: if home price is the sum of the value of the land and the value of the building, and each of those are a linear function of their square footage. I'll evaluate which is more effective later.

```
kc = read.csv("kc_house_data.csv")
kc$logprice = log(kc$price)
ggplot() +
  geom_density(data = kc, aes(x = price))
```




```
ggplot() +
  geom_density(data = kc, aes(x = logprice))
```



First, I'll ensure the columns are properly typed and remove irrelevant columns.

```
kc$id = NULL
kc$date = NULL #Since all sales are within a year (May 2014-May 2015), I assume they are in constant do
kc$zipcode = as.factor(kc$zipcode)
sum(kc$sqft_living - (kc$sqft_above + kc$sqft_basement)) #these vars are perfectly collinear, so I drop
```

```
## [1] 0
```

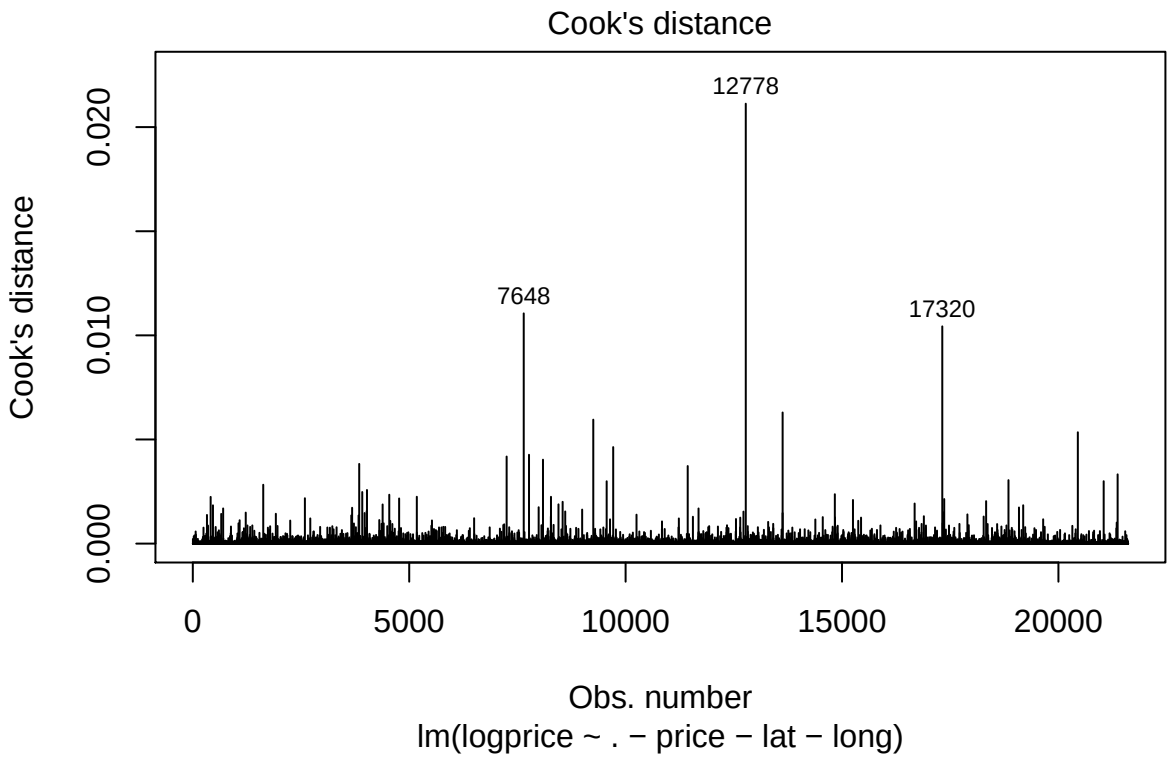
```
kc$sqft_living = NULL

#Our users probably don't have these neighborhood averages on hand, so we ignore them
kc$sqft_living15 = NULL
kc$sqft_lot15 = NULL
```

Second, I'll check for duplicate observations, missing/NA values, and remove extreme outliers unrepresentative of the relationship we're estimating.

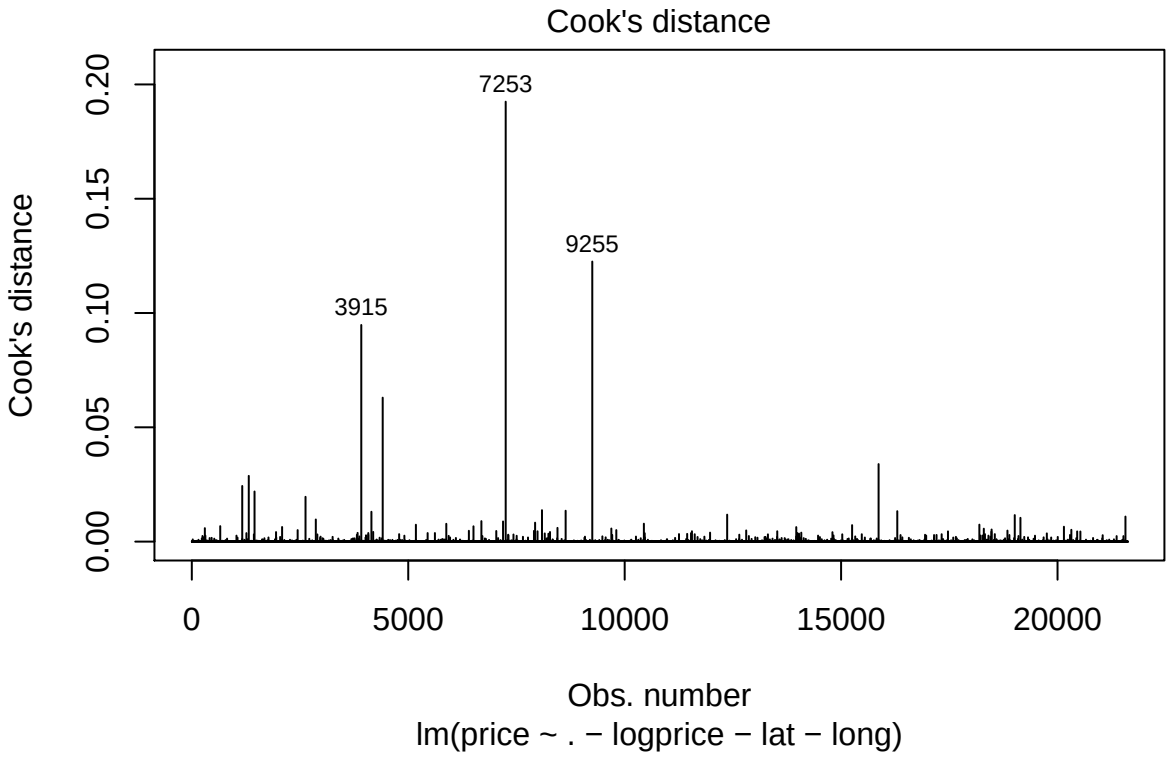
```
kc = kc[!duplicated(kc),] #there were 5 duplicated rows
sum(is.na(kc)) #there are no NA values
```

```
## [1] 0
```

```
#outliers in log-lin space
lm = lm(logprice ~ .-price-lat-long, data = kc)
plot(lm, 4)
```




```
#kc[15871,] #probably a typo - this should be 3 bedrooms, not 33
kc[15871,"bedrooms"] = 3
#kc["12778",] #this obs is a really expensive horse farm
#kc["7648",] #this obs is an extremely large lot
#kc["17320",] #this obs is again, a very large lot for a cheap price
kc = kc[!rownames(kc) %in% c("12778","7648","17320"), ]

#outliers in lin-lin space
lm = lm(price ~ .-logprice-lat-long, data = kc)
plot(lm, 4)
```




```
kc = kc[!rownames(kc) %in% c("7253","9255","3915","4412"), ]
```

Baseline Model: Naive OLS

Now that we have a good-quality dataset to work with, I want to see how a simple linear model performs using the provided variables (before we do any feature engineering). We can then use this standard to assess any future improvements.

I use 10-fold cross-validation to get a good estimate of the test error.

```
#split data into 10 folds
kc  = kc[sample(nrow(kc)),] #shuffle data
folds = cut(seq(1,nrow(kc)),breaks=10,labels=FALSE) #stores the assigned fold for each observation

#helper function to return test RMSEs and in-sample R2s for the 10 splits, along with the last lm
runOLS = function(my_formula) {
  formula = as.formula(my_formula)
  rmses = NULL
  r2s = NULL
  lm = NULL
  for(i in 1:10){
      testIndexes <- which(folds==i,arr.ind=TRUE)
      test = kc[testIndexes, ]
      train = kc[-testIndexes, ]
      lm = lm(formula, data = train)
      test_pred = predict(lm, newdata = test)
      if (substring(my_formula,1,3)=="log") {
        mse = sum((exp(test_pred) - exp(test$logprice))^2)/length(test_pred)
      } else {
        mse = sum((test_pred - test$price)^2)/length(test_pred)
```

```
      }
      rmse = sqrt(mse)
      rmses[i] = rmse
      r2s[i] = summary(lm)$r.squared
  }
  return(list(rmses,r2s,lm))
}
```

The performance of the simple log-linear model:

```
results = runOLS("logprice ~ .-price-lat-long")
paste0("Test RMSE: $",mean(results[[1]]))
```

```
## [1] "Test RMSE: $145505.493522127"
```

```
paste0("Training R-squared: ",mean(results[[2]]))
```

```
## [1] "Training R-squared: 0.873704233516618"
```

I'm also going to add these results to a dataframe to keep track of the performance of all of our models.

```
model_performance = data.frame("model" = rep("naive log-lin",10), "geo" = rep("zipcode",10), "rmse" = r
```

And the linear-linear model:

```
results = runOLS("price ~ .-logprice-lat-long")
paste0("Test RMSE: $",mean(results[[1]]))
```

```
## [1] "Test RMSE: $154500.869945318"
```

```
paste0("Training R-squared: ",mean(results[[2]]))
```

```
## [1] "Training R-squared: 0.814425810026202"
```

```
model_performance = rbind(model_performance, data.frame("model" = rep("naive lin-lin",10), "geo" = rep(
```

So far, the log-linear model seems more performant.

Feature Engineering

Polynomials & Interaction Terms

I think we can improve on model performance by creating a bunch of polynomial variables and interaction effects to capture possible nonlinear relationships and joint effects. Not all of these will be useful predictors – later, we'll regularize our model with LASSO to reduce our specification.

```
kc$bedrooms_2 = kc$bedrooms^2
kc$bedrooms_3 = kc$bedrooms^3
kc$bathrooms_2 = kc$bathrooms^2
kc$bathrooms_3 = kc$bathrooms^3
kc$bed.bath = kc$bedrooms * kc$bathrooms
kc$floors_2 = kc$floors^2
kc$floors_3 = kc$floors^3

kc$sqft_lot_2 = kc$sqft_lot^2
kc$sqft_lot_3 = kc$sqft_lot^3
kc$sqft_above_2 = kc$sqft_above^2
kc$sqft_above_3 = kc$sqft_above^3
kc$sqft_basement_2 = kc$sqft_basement^2
```

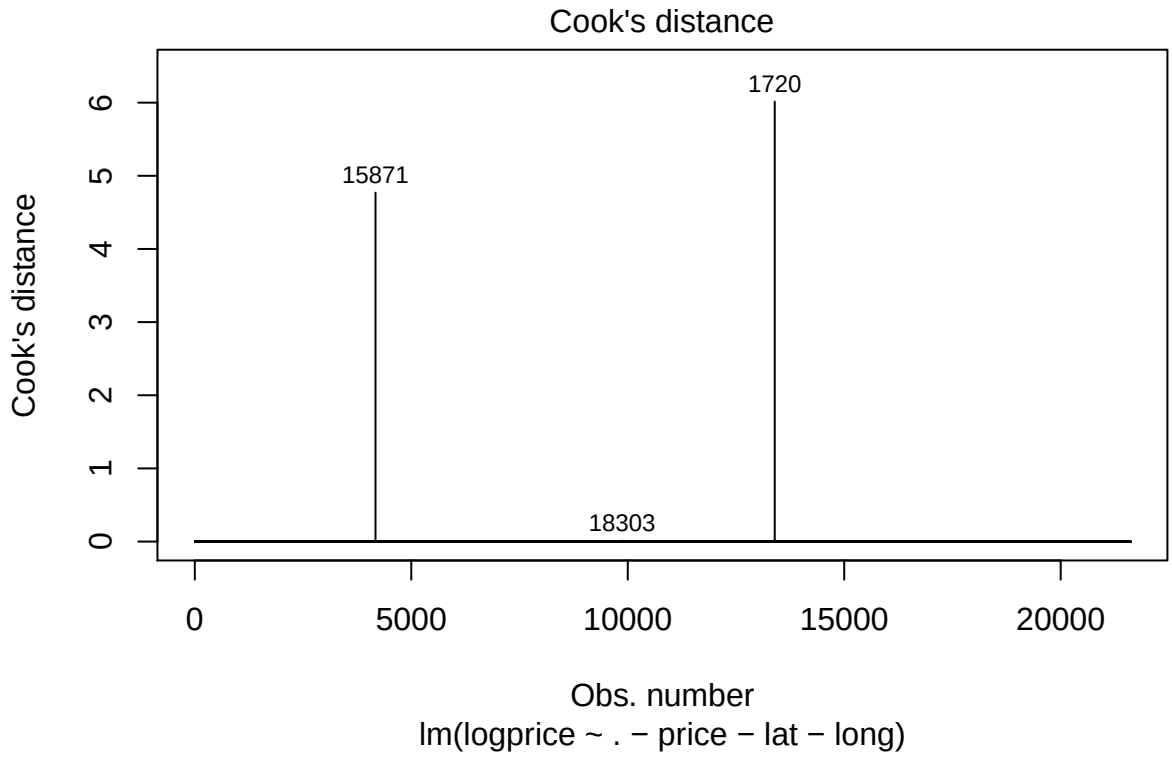
```
kc$sqft_basement_3 = kc$sqft_basement^3

kc$basement = replace(kc$sqft_basement, kc$sqft_basement>0, 1)
kc$view.waterfront = kc$waterfront * kc$view
kc$view_2 =  kc$view^2
kc$condition_2 =  kc$condition^2
kc$grade_2 = kc$grade^2

kc$age = 2015 - kc$yr_built
kc$condition.age = kc$condition * kc$age
kc$grade.age = kc$grade * kc$age
kc$age_2 = kc$age^2
kc$age_3 = kc$age^3
kc$age_since_ren_or_build = 2015 - pmax(kc$yr_built,kc$yr_renovated)
kc$age_since_ren_or_build_2 = kc$age_since_ren_or_build^2
kc$renovated = replace(kc$yr_renovated, kc$yr_renovated>0, 1)
kc$renovated.age2 = kc$renovated * kc$age^2
```

I'm dropping yr_built and $yr_renovated$ since we've converted those into more meaningful variables.

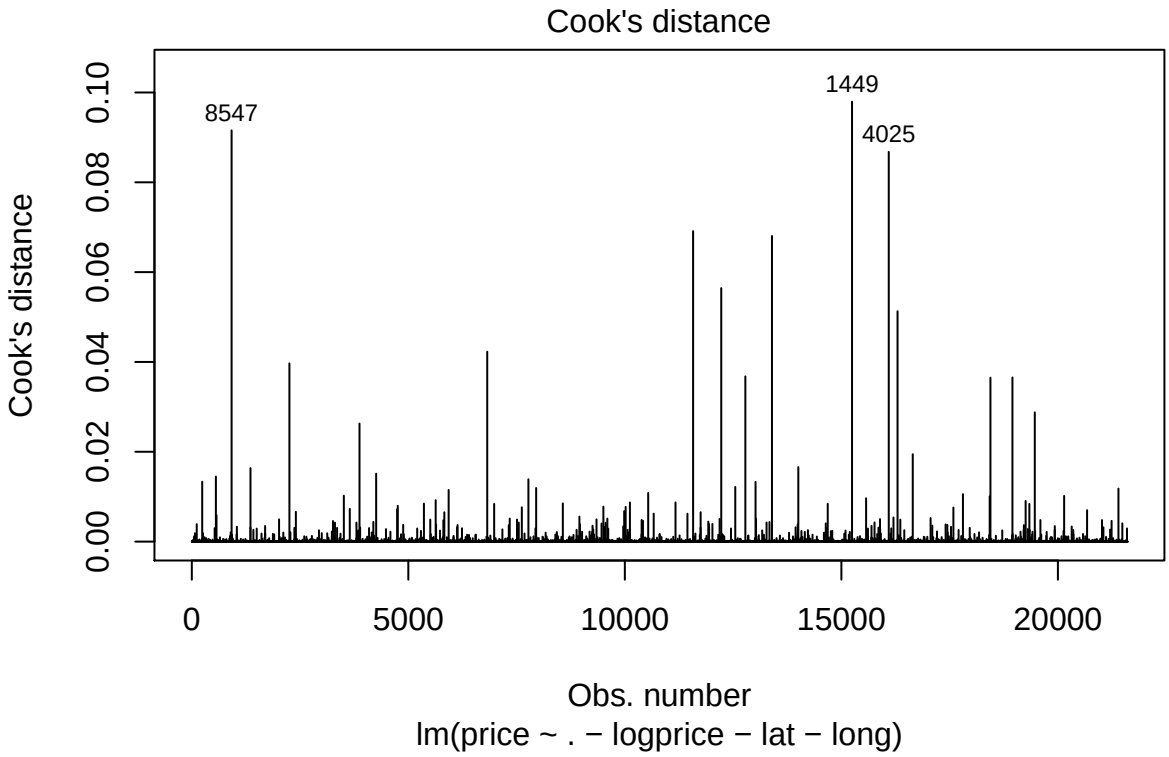
```
kc$yr_built = NULL
kc$yr_renovated = NULL
```

Since we've changed our data pretty substantially, we check once more for outlying observations.

```
lm = lm(logprice ~ .-price-lat-long, data = kc)
plot(lm, 4) #there are some pretty dangerous outliers
```



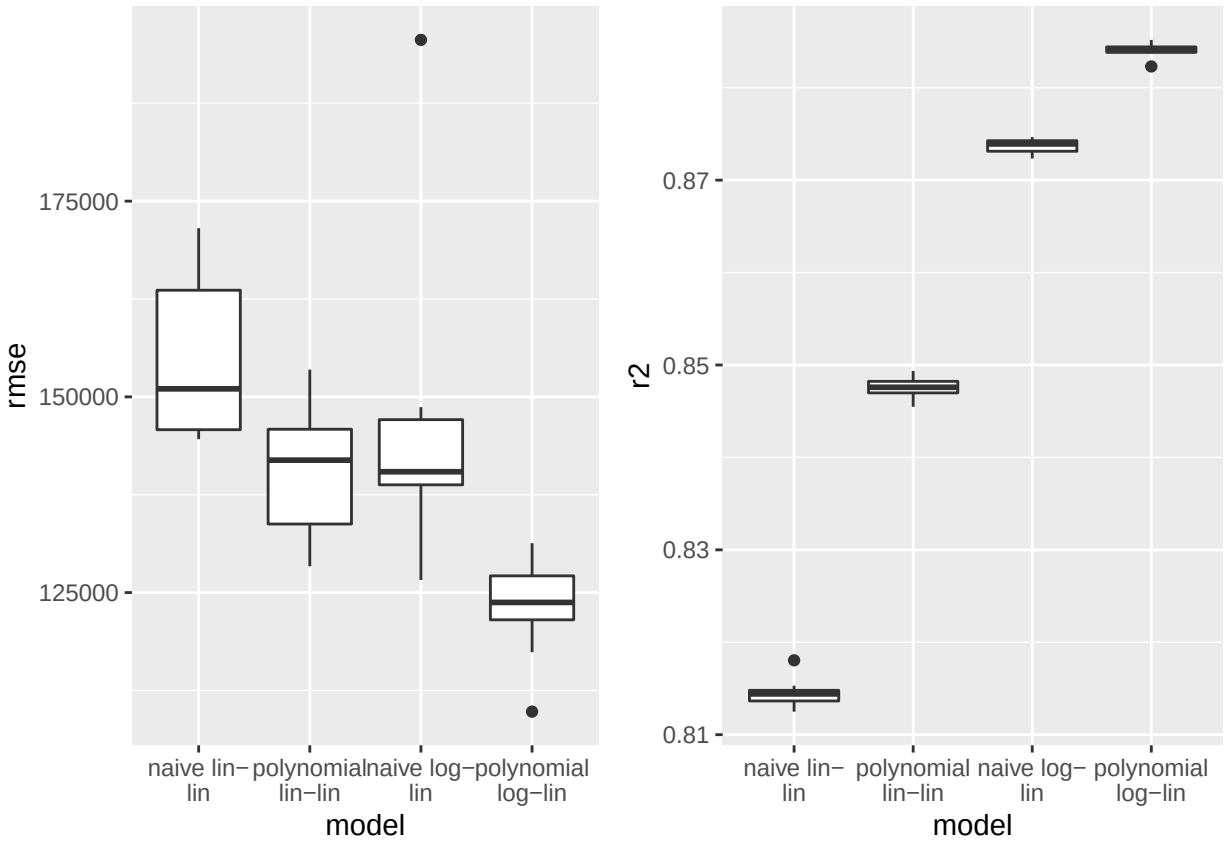

```
kc = kc[!rownames(kc) %in% c("1720","15871","18303","7770","8758"), ] #that I remove
lm = lm(price ~ .-logprice-lat-long, data = kc)
plot(lm, 4) #does not reveal any major outliers in lin-lin space.
```




```
#meaning we need to recalculate folds
kc   = kc[sample(nrow(kc)),]
folds = cut(seq(1,nrow(kc)),breaks=10,labels=FALSE)
```

Results

```
results = runOLS("logprice ~ .-price-lat-long")
model_performance = rbind(model_performance, data.frame("model" = rep("polynomial log-lin",10), "geo" =

results = runOLS("price ~ .-logprice-lat-long")
model_performance = rbind(model_performance, data.frame("model" = rep("polynomial lin-lin",10), "geo" =

model_performance$model = factor(model_performance$model, levels = c("naive lin-lin", "polynomial lin-li

p1 = ggplot(model_performance, aes(x=model, y=rmse)) +
  geom_boxplot() +
  scale_x_discrete(labels = function(x) str_wrap(x, width = 10))
p2 = ggplot(model_performance, aes(x=model, y=r2)) +
  geom_boxplot() +
  scale_x_discrete(labels = function(x) str_wrap(x, width = 10))
grid.arrange(p1, p2, ncol=2)
```



These transformations have substantially reduced our test error and improved our in-sample R^2 for both the log-lin and lin-lin models. The log-lin specification continues to perform better.

Location

Our dataset has two variables to describe a home's location: *zipcode* and *lat/long*. In the naive model used above, we regressed on *zipcode* as a factor and ignored *lat/long*. But zipcode draws arbitrary boundaries that don't necessarily reflect real-world neighborhoods. On the other hand, zipcode may effect home prices if school boundaries are partially defined using zip codes.

K-means Clustering

I use the k-means clustering approach to group houses into k groups according to geographic proximity, creating a new "neighborhood" variable. Here, I use euclidian distance to measure the distance between two lat/longs – while haversine distance would technically be more accurate, the two approaches are basically the same given our regional scale.

We do need to choose the appropriate level of granularity for "neighborhood" – for illustrative purposes, here is what 3, 10, and 100 neighborhoods look like. For comparison, there are 70 zip codes.

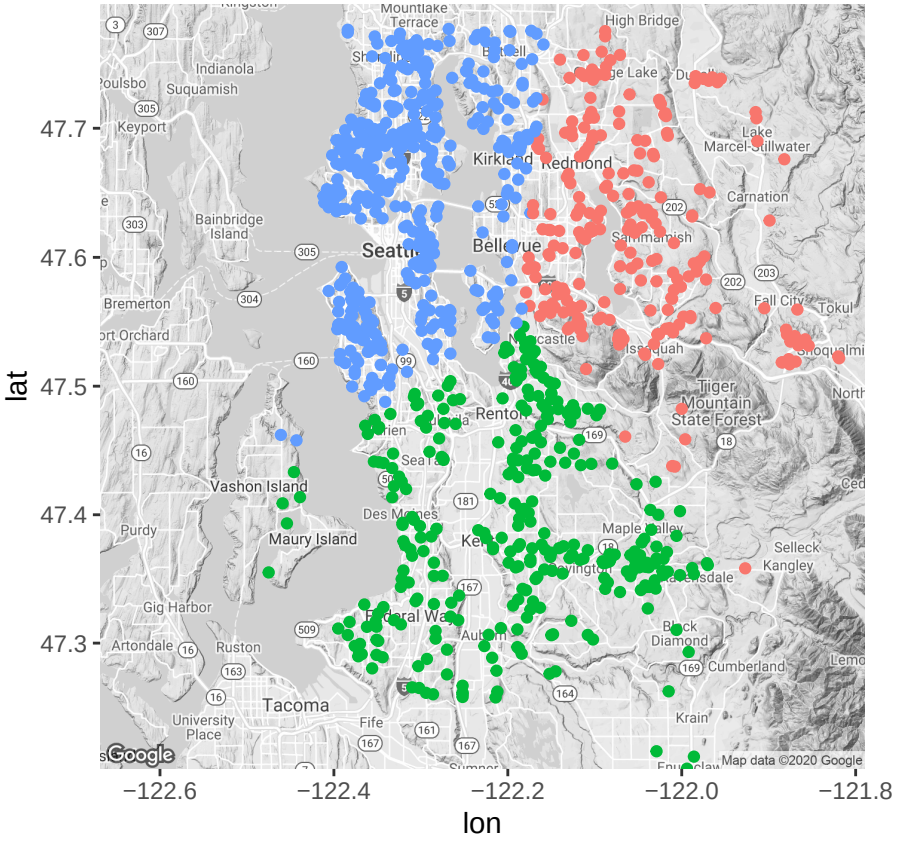
```
google_key = as.character(read.delim("my_key.txt", header = FALSE)[1,]) #insert your own google key her
register_google(key = google_key)
map = get_map(location=c(lon = -122.23, lat = 47.5), zoom=10, maptype = "terrain", source='google', col
```

```
## Source : https://maps.googleapis.com/maps/api/staticmap?center=47.5,-122.23&zoom=10&size=640x640&scal
```

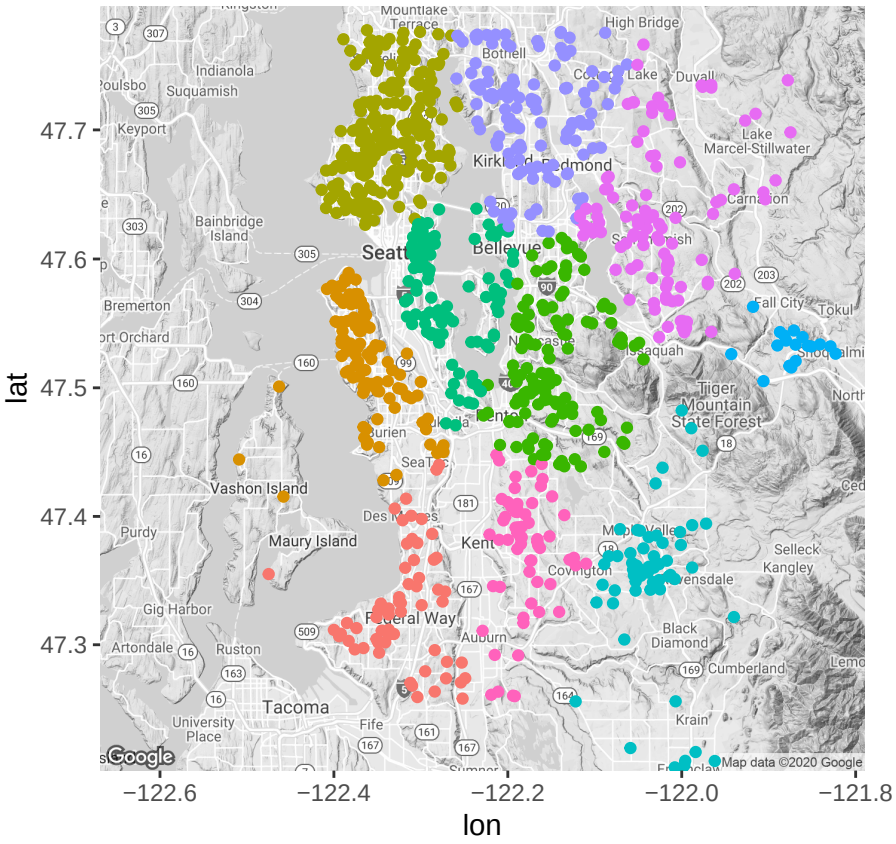
```
n = c(3,10,100)
for (i in c(1:3)){
  #generate neighborhood variable
```

```
  km = kmeans(kc[c("lat","long")],n[i])
  kc$neighborhood = as.factor(km$cluster)

  #sample 1000 points
  sample_index = sort(sample(seq_len(nrow(kc)), replace = FALSE, size = 1000))
  sample = kc[sample_index,c("lat","long","neighborhood")]

  #plot
  plot = ggmap(map) + geom_point(data = sample, aes(x=long, y=lat, color=neighborhood)) + theme(legend.p
  print(plot)
}
```

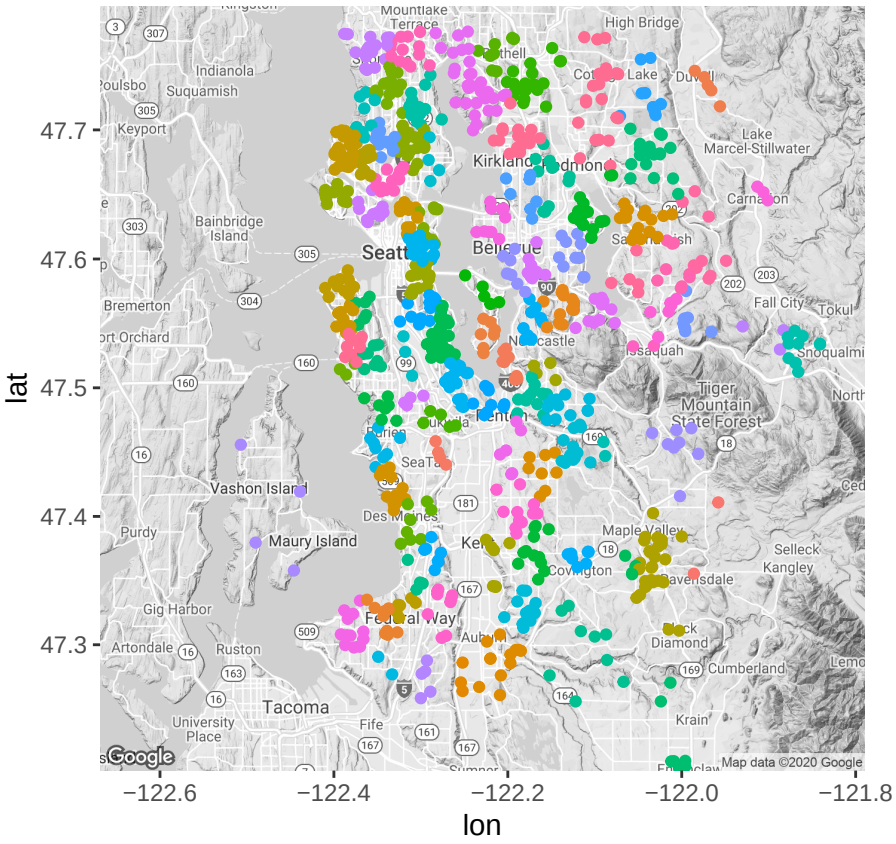
```
## Warning: Removed 9 rows containing missing values (geom_point).
```




```
## Warning: Removed 17 rows containing missing values (geom_point).
```




```
## Warning: Removed 15 rows containing missing values (geom_point).
```



To find the optimal number of neighborhoods, I search a coarse grid between 3 and 200 and pick the number that minimizes the model's test set performance.

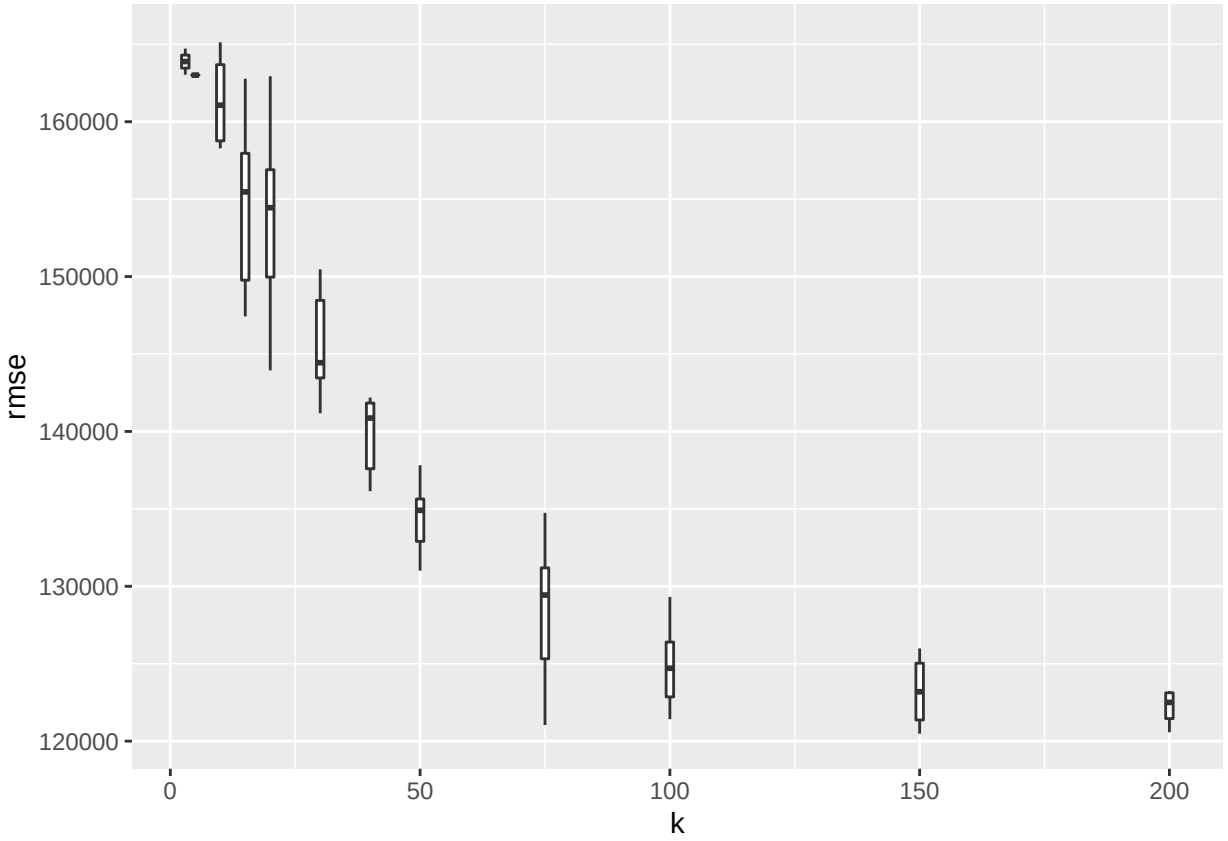
```
bootstrap_results = NULL
for (j in c(1:10)) {
  train_index = sort(sample(seq_len(nrow(kc)), replace = FALSE, size = floor(nrow(kc)*.8)))
  grid = c(3,5,10,15,20,30,40,50,75,100,150,200)
  grid = data.frame("k" = grid, "rmse" = rep(NA, length(grid)))
  for (i in 1:nrow(grid)) {
    km = kmeans(kc[c("lat","long")],grid[i,"k"])
    kc$neighborhood = as.factor(km$cluster)
    lm = lm(logprice ~ .-price-zipcode-lat-long, data = kc[train_index,])
    test_pred = predict(lm, newdata = kc[-train_index,])
    mse = sum((exp(test_pred) - exp(kc[-train_index,"logprice"]))^2)/length(test_pred)
    rmse = sqrt(mse)
    grid[i,"rmse"] = rmse
  }
  bootstrap_results = rbind(bootstrap_results,grid)
}
```

```
## Warning: did not converge in 10 iterations

## Warning: did not converge in 10 iterations

## Warning: did not converge in 10 iterations
```

```
bootstrap_results$k_fac = as.factor(bootstrap_results$k)
ggplot(bootstrap_results, aes(x=k, y=rmse,group=k_fac))+
```

```
  geom_boxplot(outlier.shape = NA) +
  scale_y_continuous(limits = quantile(bootstrap_results$rmse, c(0.1, 0.8)))
```

```
## Warning: Removed 36 rows containing non-finite values (stat_boxplot).
```



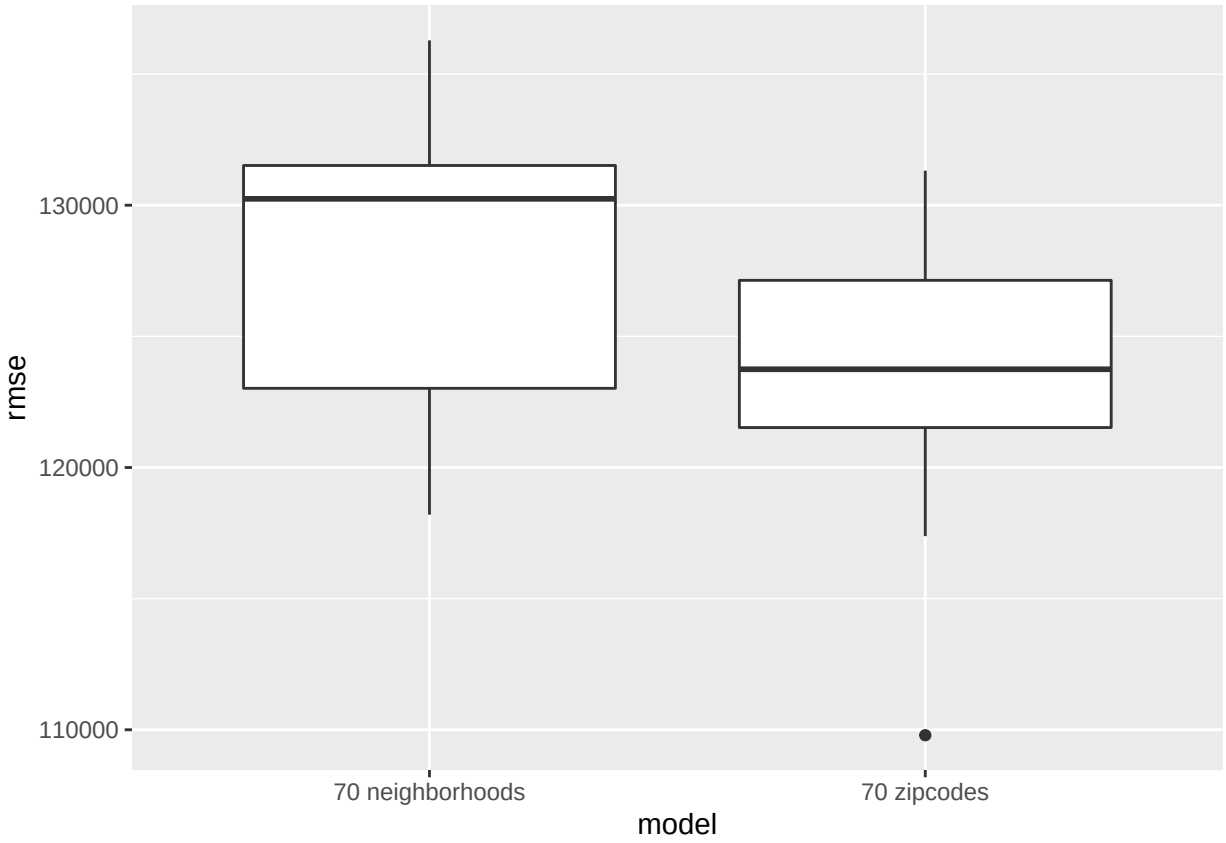
As we increase the number of neighborhoods from 1, I'd expect the prediction error to first fall (as the more flexible model reduces bias) and then rise (as we introduce variance by overfitting to the training data). However, the prediction error generally falls throughout the range of values we've tested, with a slight uptick at 200 neighborhoods. I suspect that much higher numbers of neighborhoods do create significant overfit (15-20K neighborhoods, for example, would fit every training observation about perfectly).

Using K-Means Neighborhood Clusters in Conjunction with Postal Codes

So far, we've just looked at geographic clustering in isolation. But is it more or less effective than using zipcode?

```
kc$neighborhood = NULL
results = runOLS("logprice~.-price-lat-long")
results.zipcode = results[[1]]

km = kmeans(kc[c("lat","long")],70)
kc$neighborhood = as.factor(km$cluster)
results = runOLS("logprice~.-price-lat-long-zipcode")
results.neighborhood = results[[1]]

comparison = data.frame("model" = c(rep("70 zipcodes",10),rep("70 neighborhoods",10)), rmse = c(results

ggplot(comparison, aes(x=model, y=rmse)) +
```

```
  geom_boxplot()
```



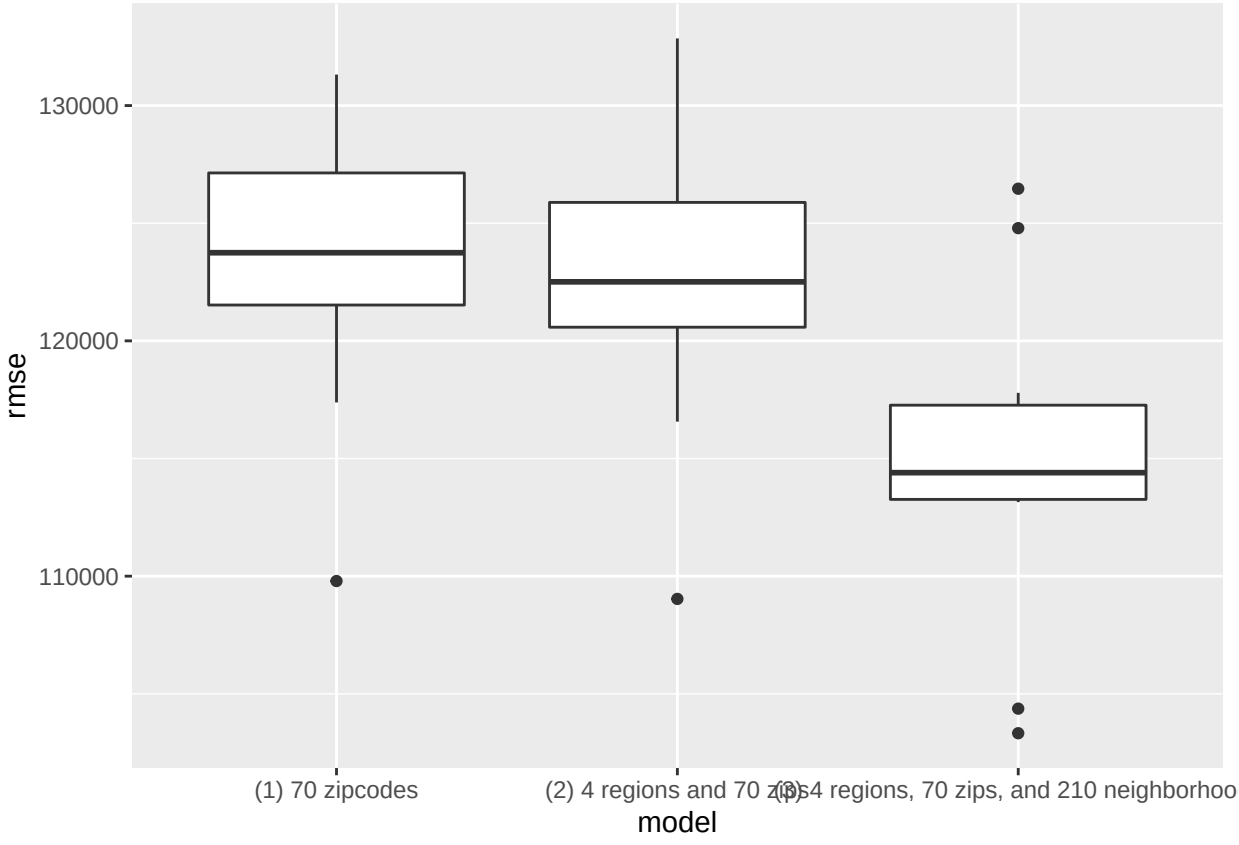
In a head-to-head matchup, our model seems to perform better using 70 postal codes compared to 70 neighborhoods - zip code matters more than geographical proximity! What if we use a much more granular neighborhood measure in conjunction with postal code?

```
kc$neighborhood = NULL
results = runOLS("logprice~.-price-lat-long")
results.zipcode = results[[1]]

km = kmeans(kc[c("lat","long")],4)
kc$region = as.factor(km$cluster)
results = runOLS("logprice~.-price-lat-long")
results.region = results[[1]]

km = kmeans(kc[c("lat","long")],210)
kc$neighborhood = as.factor(km$cluster)
results = runOLS("logprice~.-price-lat-long")
```

```
## Warning in predict.lm(lm, newdata = test): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(lm, newdata = test): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(lm, newdata = test): prediction from a rank-deficient fit
## may be misleading
```

```
## Warning in predict.lm(lm, newdata = test): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(lm, newdata = test): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(lm, newdata = test): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(lm, newdata = test): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(lm, newdata = test): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(lm, newdata = test): prediction from a rank-deficient fit
## may be misleading

## Warning in predict.lm(lm, newdata = test): prediction from a rank-deficient fit
## may be misleading
```

```
results.neighborhood = results[[1]]

comparison = data.frame("model" = c(rep("(1) 70 zipcodes",10),rep("(2) 4 regions and 70 zips",10), rep(

ggplot(comparison, aes(x=model, y=rmse)) +
  geom_boxplot()
```

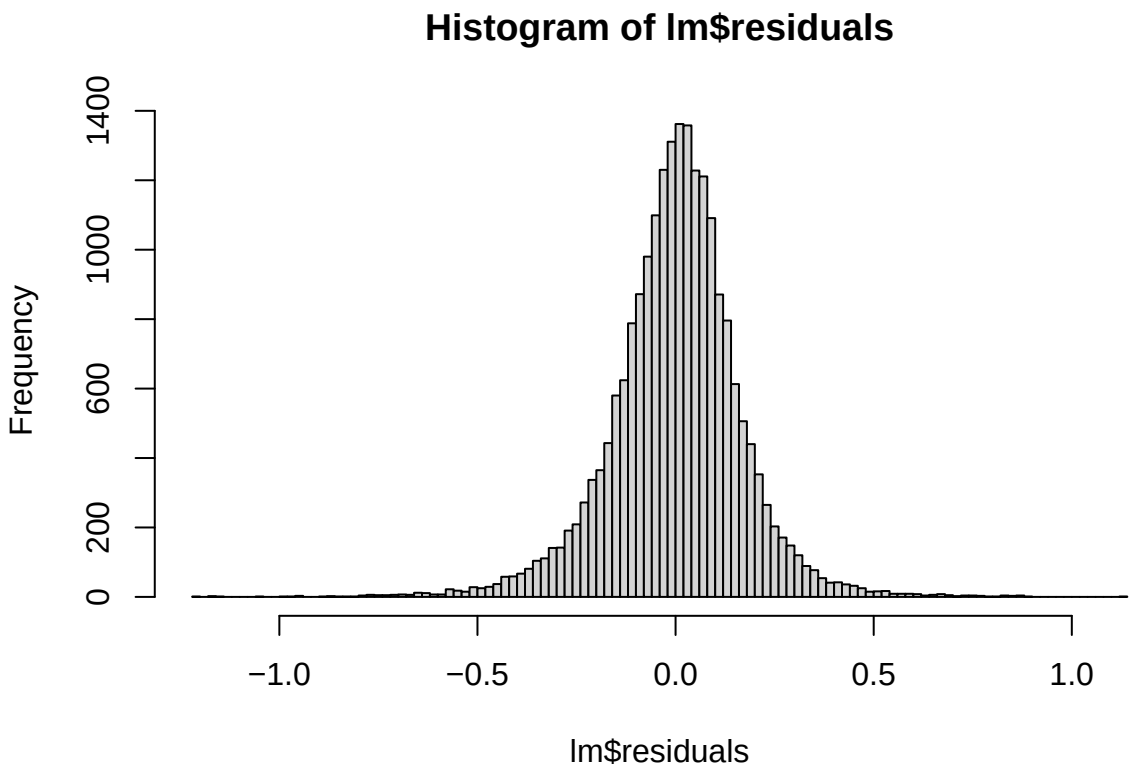


These results suggest that adding the more granular neighborhood measure helps the model's performance significantly.

Testing OLS Assumptions

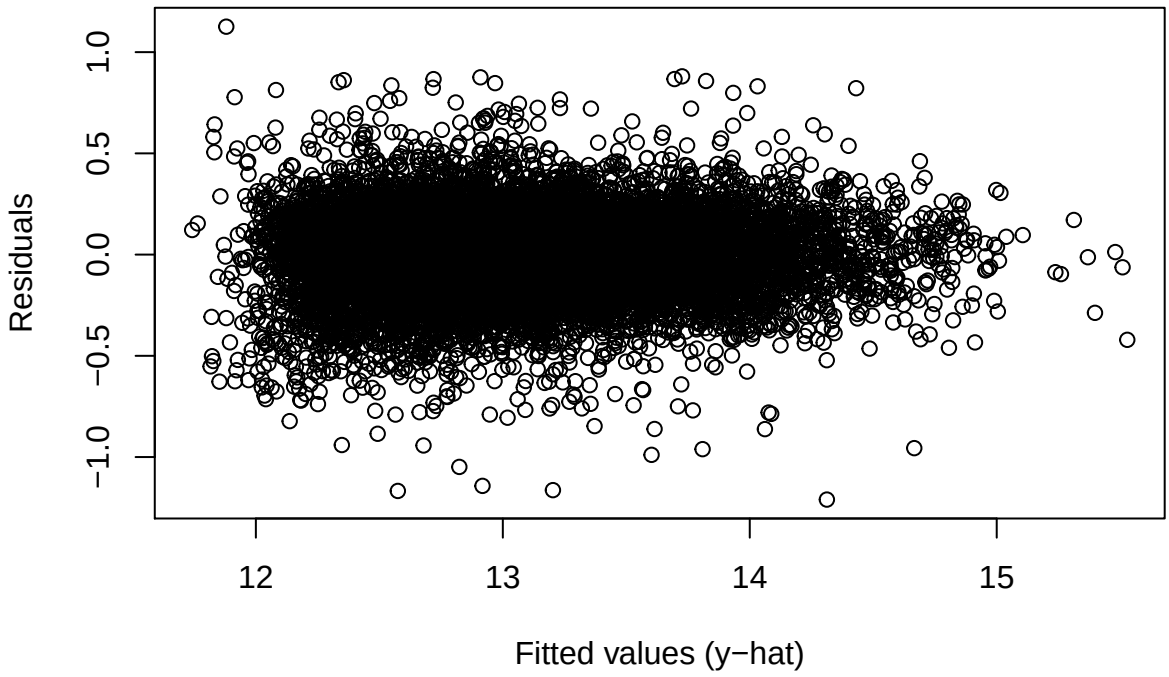
Residuals are normally distributed:

```
lm = lm(logprice ~ .-lat-long-price, data = kc)
hist(lm$residuals, breaks = 100)
```



Data appears homoskedastic:

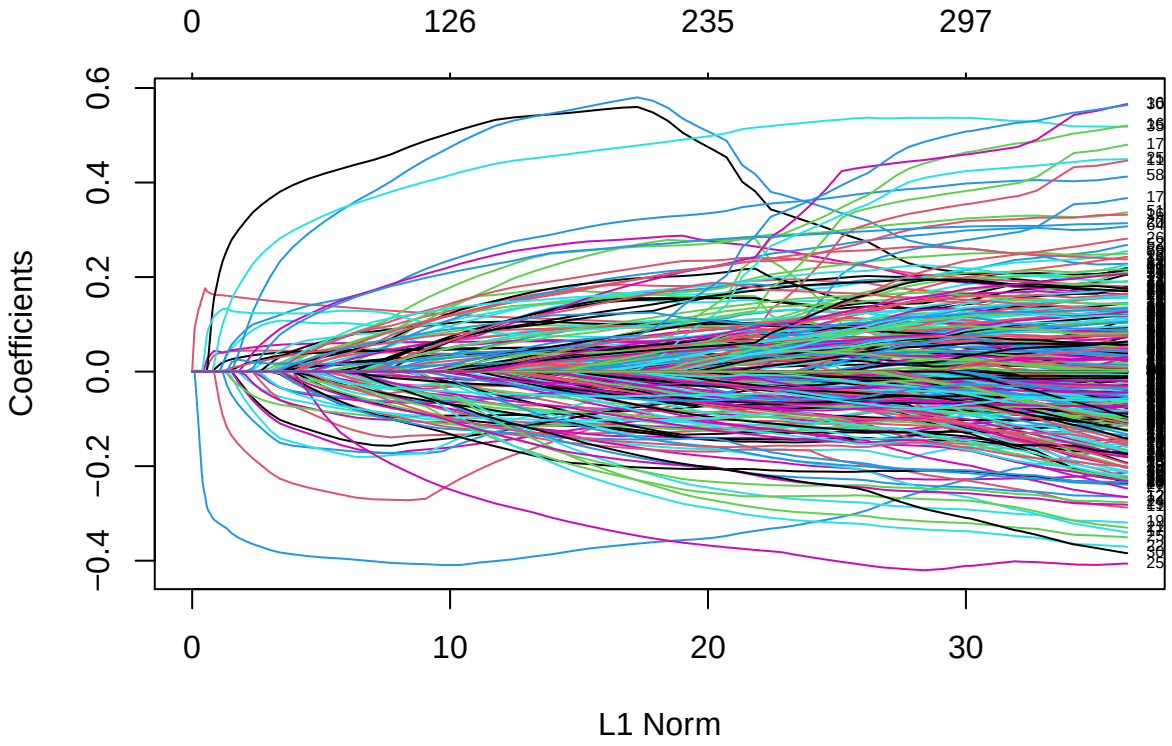
```
plot(lm$fitted.values, lm$residuals,
     xlab = "Fitted values (y-hat)",
     ylab = "Residuals")
```



Variable Regularization

To build our final model, I use LASSO to see if we can get any improvement in our test MSE through regularization.

```
x = model.matrix(logprice ~ .-price-lat-long, data = kc)[,-1]
y = kc$logprice

train_index = sort(sample(seq_len(nrow(kc)), replace = FALSE, size = floor(nrow(kc)*.8)))
lasso_model = glmnet(x[train_index,],y[train_index],alpha=1) #performs normalization by default
plot(lasso_model, label = TRUE)
```



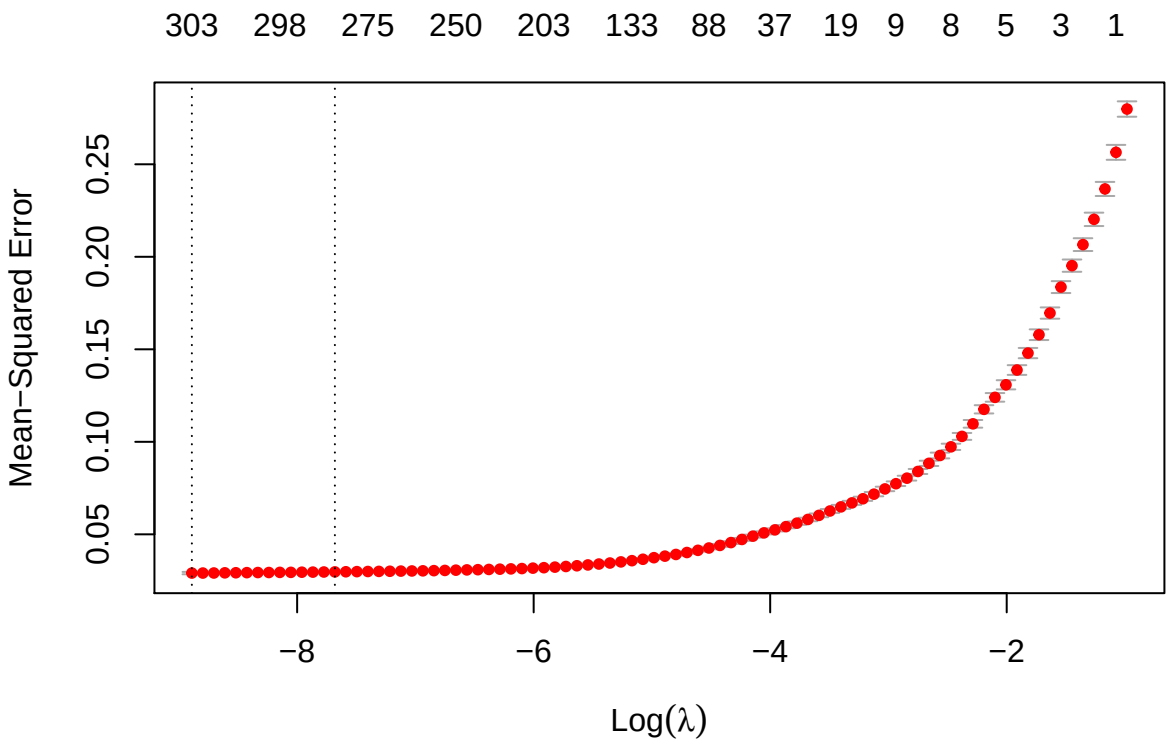

```
#whats in a sparse model?
lasso_model$lambda[13]
```

```
## [1] 0.1225747
```

```
lasso.coef = coef(lasso_model)[,13]
lasso.coef[lasso.coef!=0]
```

```
##   (Intercept)         grade    sqft_above      bed.bath       grade_2
##  1.174296e+01  1.535141e-01  3.421629e-05  2.469975e-03  1.697531e-03
##       region3
## -2.168933e-01
```

```
#use CV to find the optimal lambda
cv.out=cv.glmnet(x[train_index,],y[train_index],alpha=1)
bestlam=cv.out$lambda.min
plot(cv.out)
```




```
#estimate the out-of-sample RMSE
lasso.pred=predict(lasso_model,s=bestlam ,newx=x[-train_index,])
rmse = sqrt(sum(
  (exp(lasso.pred) - exp(y[-train_index]))^2
  )/length(lasso.pred))
```

Regularizing OLS isn't necessary here.

Final Model

Lastly, we re-run our model on the full dataset and save it for export.

```
lm = lm(logprice ~ .-price-lat-long, data = kc)
saveRDS(kc,file="../app/resources/kingcounty.rds")
saveRDS(lm,file="../app/resources/lm.rds")
```