Modern Data Mining - Final Project

Joe Farned

12/23/16

Contents

Introduction and Findings

In this project, we attempt to build a model for predicting Philadelphia real estate prices. This project is motivated by the business model of Opendoor, a San Francisco-based startup that creates liquidity in the housing markets by drastically reducing the amount of time it takes to sell a home. It enables sellers to offload a home with just a click. The company then re-sells the home for a (small) premium, after identifying easy improvements that will be significantly cost-positive. Hence, accurate prediction of home price is critical.

Opendoor is currently operating in Phoenix and Dallas, and expanding into Atlanta. Expanding into a new city generally requires a new model be built, owing to the unique properties of each market and variability of quality/availability of data. We want to see if it possible to build an effective model for Philadelphia.

We can produce fairly accurate predictions of Philadelphia home prices, probably sufficiently good enough to make home purchases without worrying about losing signficant amounts of money. For instance, the model run against the 2008 housing crisis indicates our total predicton error would be around $281,168 over 20,000 homes.

We identify signficant factors in a home's price to be the livable area, number of rooms/bedrooms/bathrooms, taxable land/building, the tax assessed market value, and a moving average of nearby sold homes.

Description of Data

Our data is obtained from an iniative called OpenDataPhilly, which compiles government-obtained statistics from Philadelphia. Our dataset is the "Property Parcels" dataset, which gives a record of around half a million properties. It contains information such as the square footage, sale price, assessed tax value, and more. Further discussion is given in the data cleaning section of the report.

Description of Problem

This problem can be broken into two parts: 1) predicting the home price for purchase and 2) identifying how to sell/improve the home to make a profit.

For 1), we want to first identify the actual market value of the home. This is a classic regression problem, since we will collect a set of variables and attempt to predict a value using them. A complicating factor of this problem is the aspect of time, which needs to be accounting for by trying a number of different variables that attempt to capture the effect of time. Once we have identified the proper price for a home, we then want to attempt to purchase the home for slightly less than this price, if possible. Knowing how much we can mark the price down is a question we probably do not have sufficient data to answer at this time. Thus, we will focus on predicting the right price for the home.

For 2), we want to both identify how we expect the home price to change from the time we purchase it, as well as improvements we can make in a short period of time (3-4 weeks) that will have a large marginal effect on the value of the home. We probably lack sufficient data to make these estimations at the moment, since few homes in the house are sold twice within the period of a month.

Since it is thus out of range to calculate expected profit, we will focus on how well we are able to predict the actual sale prices of the homes, recognizing that it is a critical part of a such a model.

Objective

We use the following metrics to judge the performance of our models:

1) the r-squared value of the regression
2) the total dollar amount our model is off for the year of 2013
3) the total dollar amount our model is off for the year of 2008

The reason we choose 2008 is that it was a particularly volatile year for housing owing the financial crisis. We choose 2013 since it was more stable and recent, and is well-represented in the data. We want to focus on a the span of year since it provides a real-world look into the window a business (especially a startup) has to prove itself. It is useful, further, to focus on two years since we have 500,000 rows, and running a prediction on fewer of these data points is better suited to our limited computational capacity.

System Design

I adopted the system design used by Opendoor as detailed in their engineering blog posts. For the system, we start by reading in a configuration file that specifies the method of regression that will be used, the variables to be regressed on, and other important parameters. An example of a configuration used by Opendoor is given here:

We then parse this file, and convert the appropriate variables in the dataframe to numerical, factors, or dates. Then, we split the data into test, training, and validation sets with a 60%, 20%, 20% proportion.

On the training split, we then train a model (the type specified by the configuration file: either linear regression or Ridge/Lasso/ElasticNet). We do not want to use a classifier since we are estimating dollar values. Next, we calculate a vector giving the difference between the predicted sale price for the training and the actual sale price.

Following this, we train a linear model on this "error" vector. This is a method used by Opendoor, which was originally suggested in Elements of Statistical Learning. This method is called Expected Prediction Error (EPE). We model is useful because it allows us to estimate how certain we are about the prediction our price model is making.

```
model_config = {
  'model_type': 'linear_regression',
  'features': [
    'living_area',
    'num_bedrooms',
    'num_bathrooms',
    'neighborhood'
  ],
  'feature_types': {
    'living_area': 'numerical',
    'num_bedrooms': 'categorical',
    'num_bathrooms': 'categorical',
    'neighborhood': 'categorical'
  },
  'model_params': {
    'l1': 3e-6,
    'l2': 3e-6,
    'loss': 'squared_loss'
  }
}
```

Figure 1: Config file example.

$$\mathrm{Err}(x_0) = \mathbb{E}[(Y - \hat{f}(x_0))^2 | X = x_0].$$

Figure 2: Expected Prediction Error (EPE)

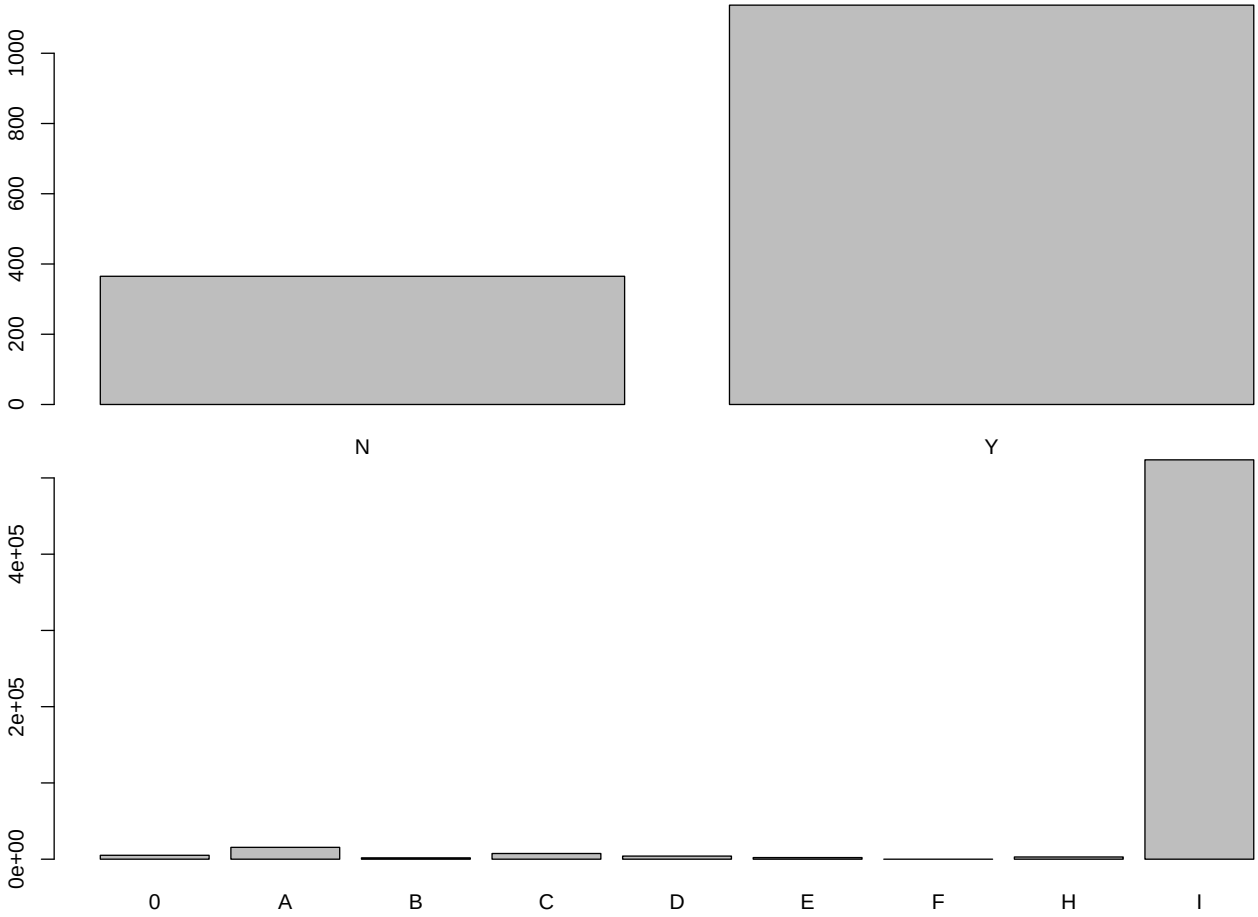
Then, on the testing data, we predict the sale price for the home, as well as the EPE. Using an ad-hoc threshold, we determine whether or not the EPE is low enough to justify purchasing the home. If so, we record it as purchased. Finally, we sum up our total error over actual sale prices.

We record our performance statistics and original configuration in a .json file stored in the "results" folder. This provides a record we can use to look back on to determine the performance a model. We find this method useful because running an "experiment" can take upwards of 20-30 minutes, and we want our results to persist to prevent having to re-run expensive computations.

Data Cleaning

We begin by taking the "Parcels" dataset and cleaning it to create something that we can use in our regression.

First, we remove columns that have incomplete ("NA") data. This step results in a removal of a large chunk of the 72 original variables. Consider, for example, the variable Other.Building. There are fewer than 1,000 yes responses and even fewer no's, which is far fewer than than half a million homes represented in the dataset. Hence, we ought to remove it.

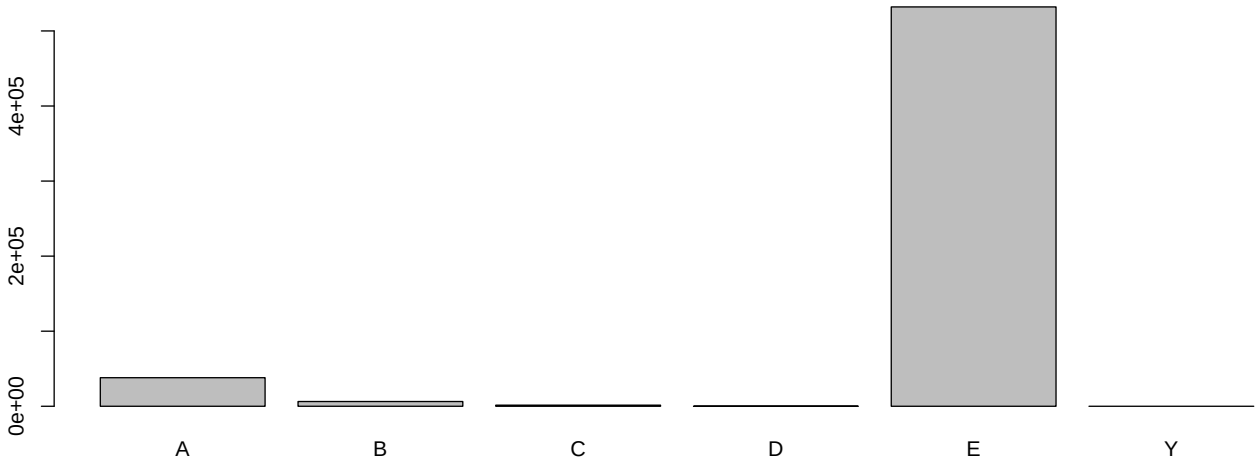


Thus, we remove: "Owner.2", "Mailing.Care.of", "Mailing.Address.1", "Mailing.Address.2", "Mailing.Street", "Mailing.City.State", "Mailing.Zip", "Street.Direction", "Site.Type", "Garage.Type", "View", "Other.Building", "General.Construction", "Date.Exterior.Condition", "Exterior.Condition", "Quality.Grade", "Basements", "Type.Heater", "Fuel", "Central.Air", "Utility", "Sewer", "Separate.Utilities", "Cross.Reference", "Unfinished"

Further, it is useful to go ahead also remove columns that we can reasonably say have no value in our regression. For example, we can reasonably say that the name of the owner (on its own) will not be a

strong predictor. Nor will Parcel or Registry number, since they are merely identifiers. These variables are "Parcel.Number", "Owner.1", "Street.Name", and "Registry.Number."

For variables such as Street.Code, we are both missing many data points, and don't have enough information to determine what the meaning of the variable is. For instance, consider Shape:



Thus, we remove "Street.Code", "Suffix", "Unit", "House.Extension", "Building.Code", "Building.Code.Description", "State.Code", "Shape", and "Book.and.Page."

Finally, we have a grouping of variables that lack complete data, but are nearly complete enough where we could impute values. For example, Year.Built is fairly consistent across the board (which may actually represent that it is inaccurate), so we may be able to predict the few missing values using a classifier. While we opt to remove them for now, we make this note for future use. These variables are "Total.Liveable.Area", "Topography", "Number.Stories", "Year.Built", "Year.Built.Estimate", and "Interior.Condition."

We remove the above variables in this step.

```
properties_clean = properties_dirty[, !(names(properties_dirty) %in% variables_out)]
```

Next, we need to convert dollar values into numerical values to save ourselves work when running regressions down the line. We do this for "Sale.Price", "Market.Value", "Taxable.Land", "Taxable.Building", "Exempt.Land", and "Exempt.Building."

It will also be good to take the coordinate tuple and convert it into numeric values:

Finally, we clean out NA values. We will have to give up on 12,196 (out of 580,181) data points to make our dataset complete. We probably do not need to worry about this since we have nearly half a million data. We do recognize the possibility that there may be some correlation between NA values, but also recognize that given the size of the dataset, this probability is low.

```
properties_clean = properties_clean[!(rowSums(is.na(properties_clean))),]
```

Finally, we save the data into a file called properties_clean.csv.

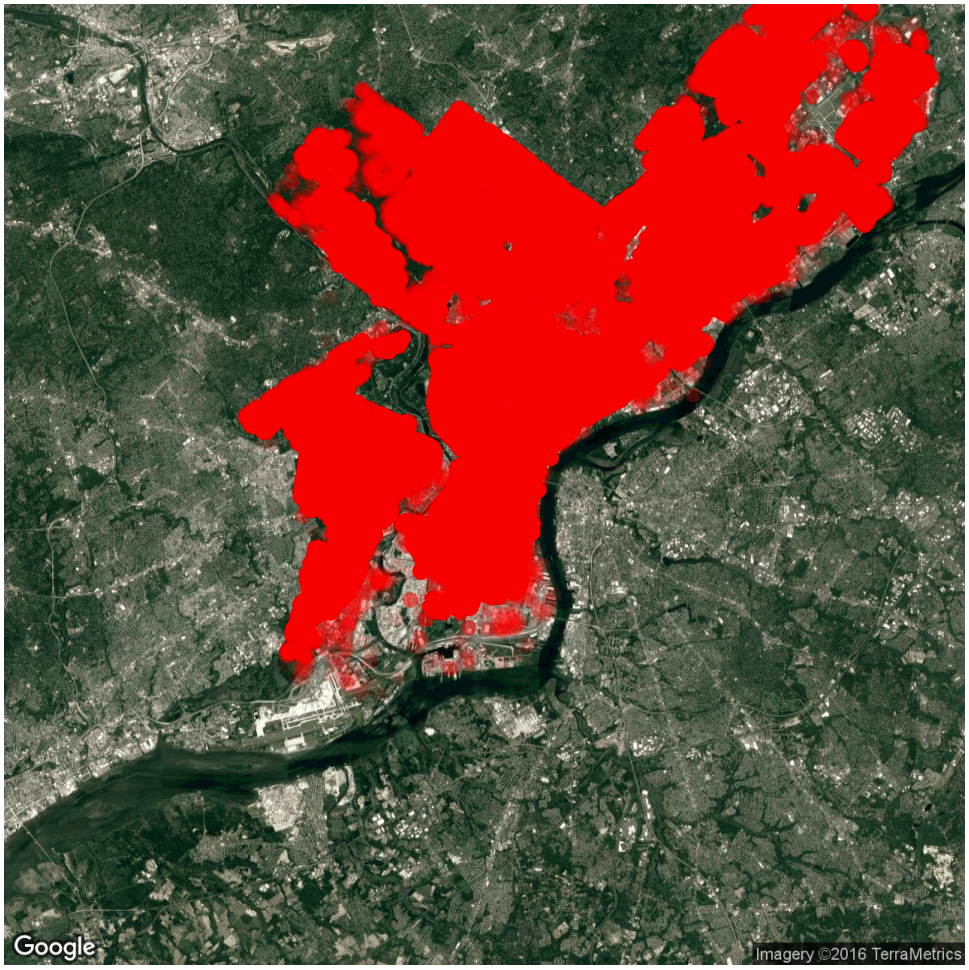
Data Exploration

In this section, we will explore the data to better understand the nature of the problem.

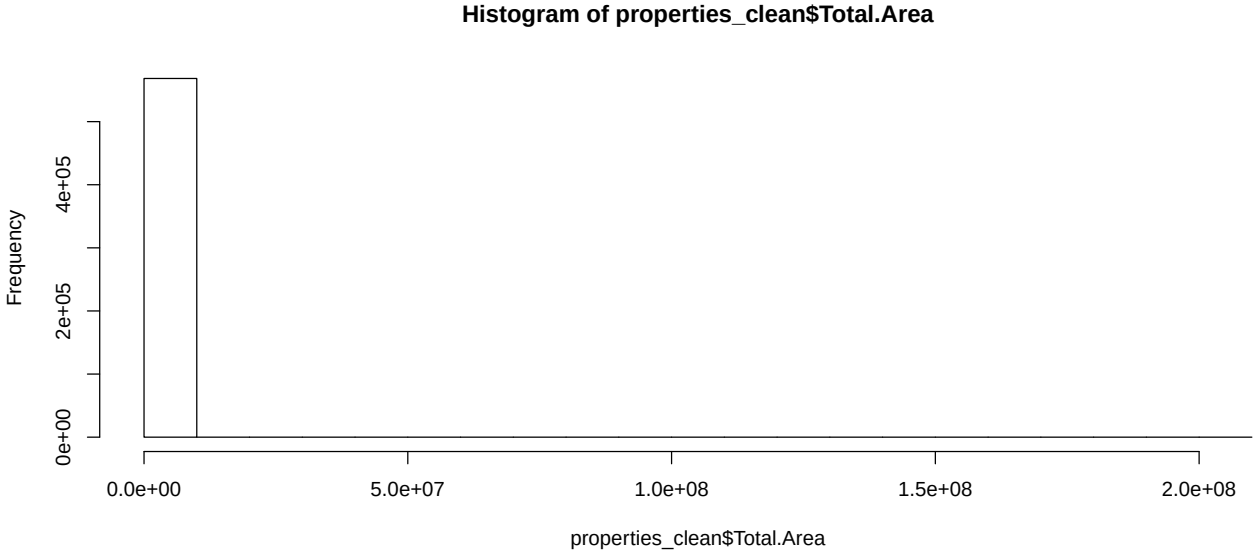
First, let's look at where the houses are located in Philadelphia.

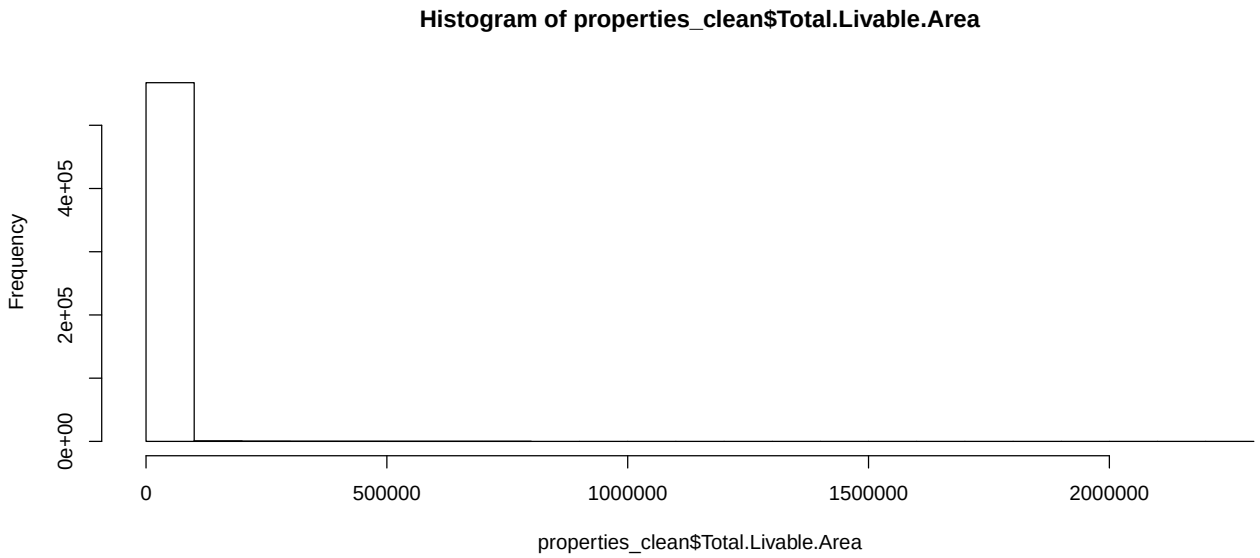
```
## Source : https://maps.googleapis.com/maps/api/staticmap?center=philadelphia&zoom=11&size=640x640&scal
```

```
## Source : https://maps.googleapis.com/maps/api/geocode/json?address=philadelphia
```

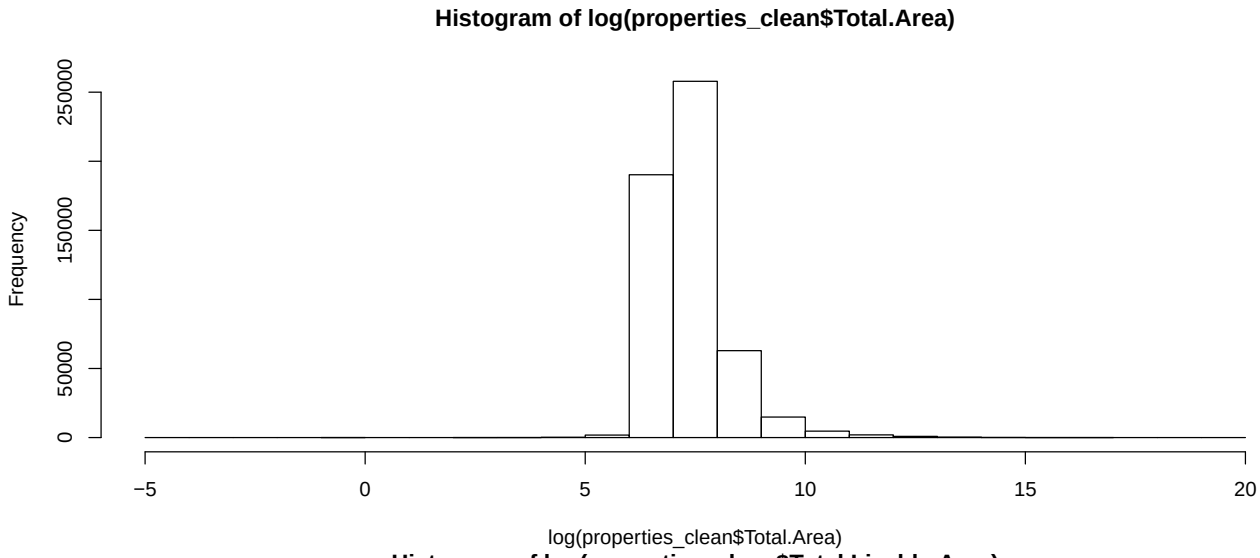


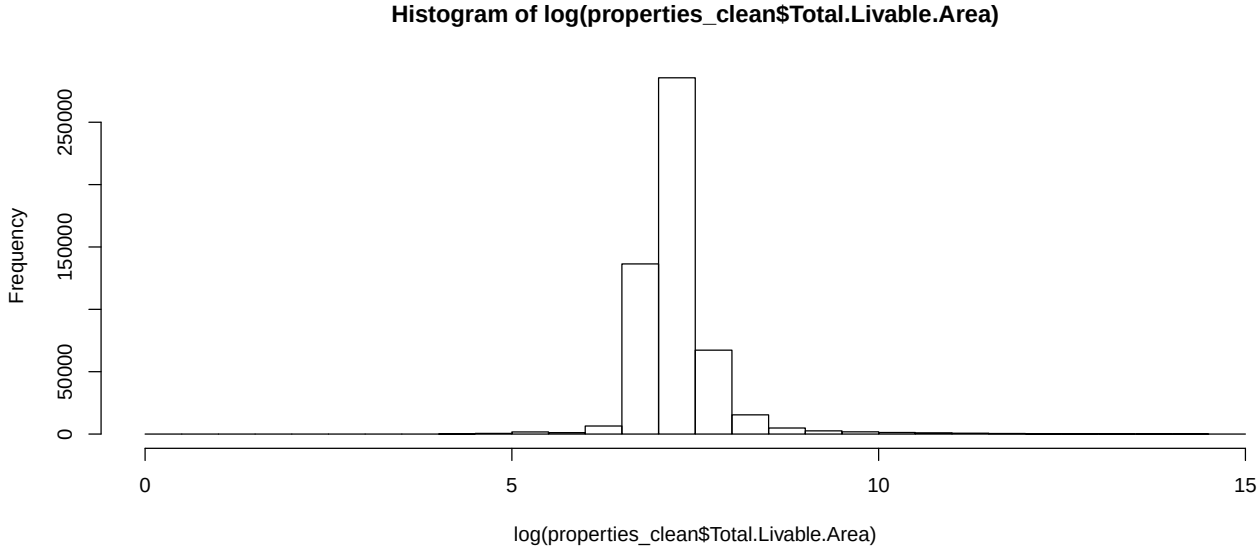
It's also likely to be pretty informative to look at the the Total.Area and Total.Livable.Area, both of which we would expect to be important predictors.



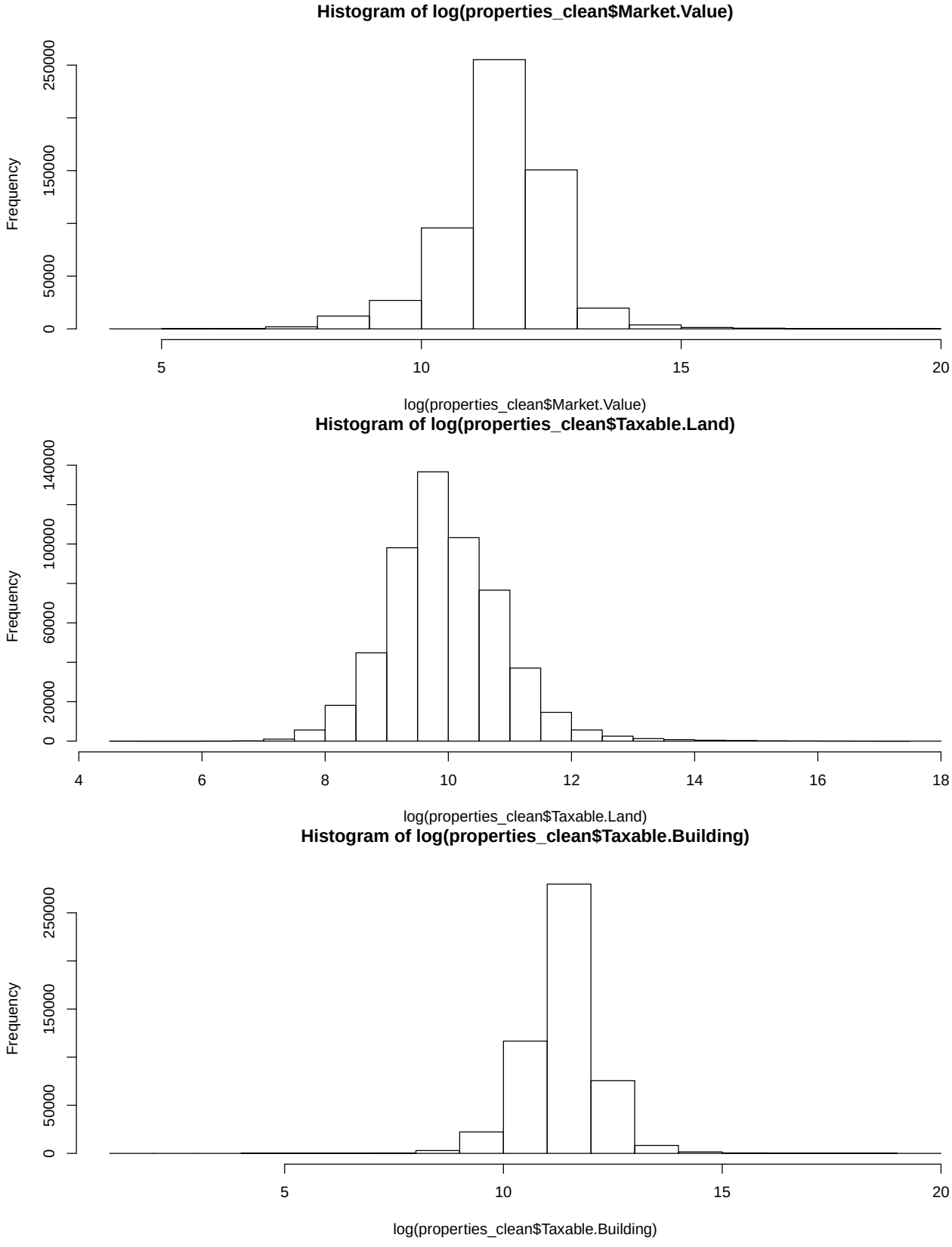


These variable looks much better after a log transformation.





Similarly, we get a much more normal distribution by taking the logs of Market.Value, Taxable.Land, Taxable.Building (only Market.Value shown).



It's also interesting to look at the top zip codes.

```
head(summary(as.factor(properties_clean$Zip.Code)))
```

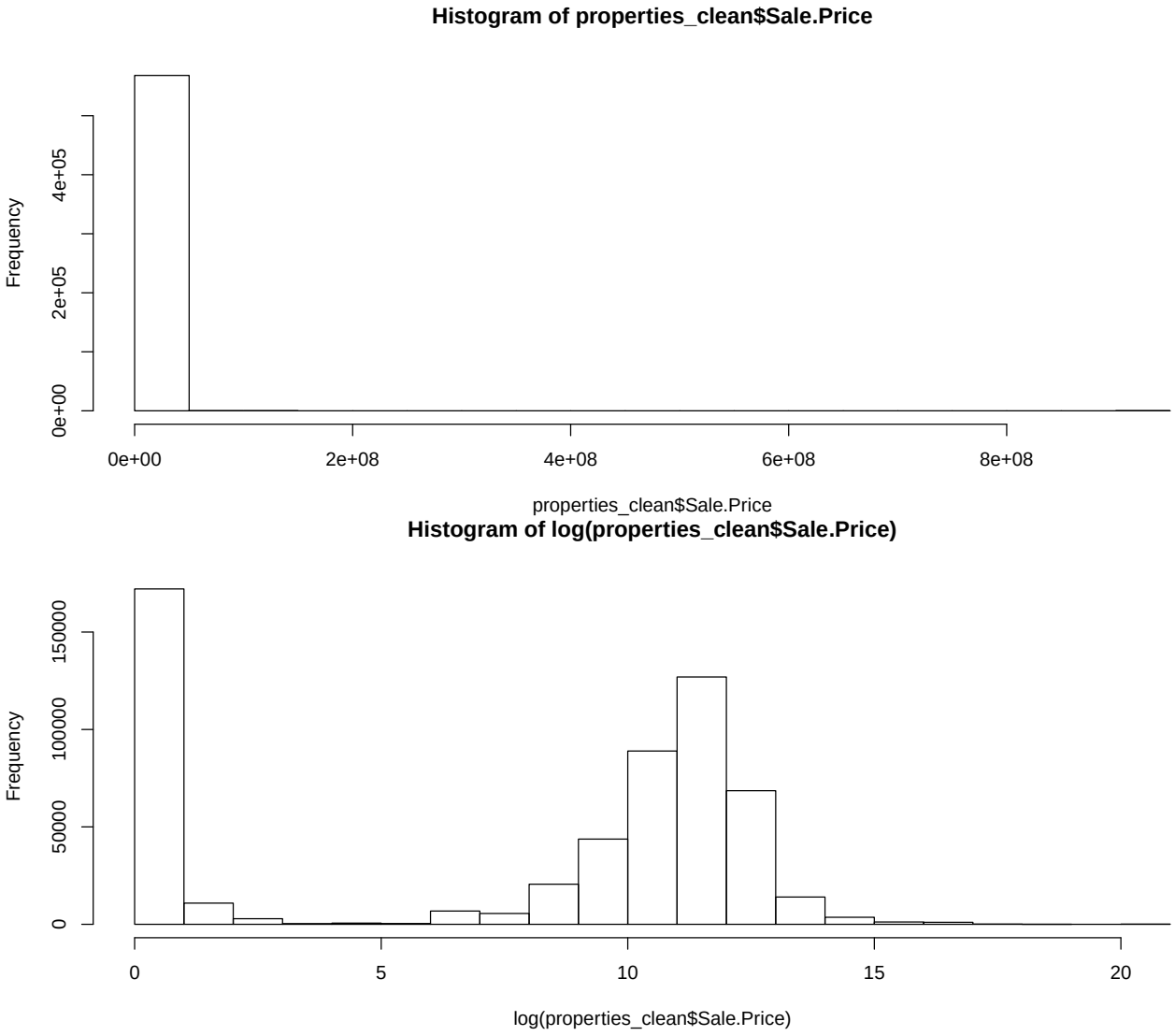
```
## 191500000 191380000 191190000 191200000 191210000 191240000
##      2312      2079      1848      1773      1301      1288
```

19150 is the most common, which is near Wyndmoor. It's followed by 19138, which is neighbors this area.

Finally, let's examine sale data. Here are the 6 dates on which the most transactions occured.

```
## 01/01/1943 12:00:00 AM 10/05/2015 12:00:00 AM 10/20/2015 12:00:00 AM
##                   1094                    625                    593
## 12/09/2015 12:00:00 AM 12/21/2005 12:00:00 AM 10/14/2015 12:00:00 AM
##                    480                    426                    410
```

As for the price,



The sale prices looks better under a log transform.

Regression

Initial linear regressions show that Zip.Code, Zoning, and Census.Tract are not useful predictors, so we remove these variables. Interestingly, these are the geography-based predictors. Further, we see that in our linear regression, Exempt.Building is "NA", so we also remove this variable.

Our system looks at the variables included at the config file and uses the "get" function R to retrieve and apply the function that we have written that will make the desired transformation. Consider Log.Total.Area, for example:

```
Log.Total.Area = function(frame, name, type) {
  frame$Log.Total.Area = log(frame$Total.Area)
  frame$Log.Total.Area = sapply(frame$Log.Total.Area, function(x) if (is.infinite(x)) { x = 0 } else {x
  return(frame)
}
```

Further, Log.Total.Area has a p-value of 0.98249835 whereas Log.Total.Livable.Area has a p-value of $<$ 2.22e-16. This is because they are highly correlated. We opt to drop Log.Total.Livable.Area.

On the formula "Sale.Price ~ Market.Value + Taxable.Land + Taxable.Building + Exempt.Land + Frontage + Depth + Number.of.Rooms + Number.of.Bedrooms + Number.of.Bathrooms + Log.Total.Livable.Area," we then obtain a r-squared of 22%. We can do better!

Let us try to add a variable that will capture the effect of nearby, recently sold houses. We will use a time-series moving average of the last 10 houses sold within 3000m to attempt to capture this effect. This variable will be called "Local.Moving.Average." To make this variable, we first need to time sort our data to reduce the runtime later down the line. This is a fairly quick computation. We then track an index running backwards in time, checking each row to see if it is within a mile; if so, we add it to the total. If there is nothing within this range, we just use the sale price of the closest home recently sold. This overall is an O(n) solution, which is far faster than the O(n^2) alternative we would face using data not sorted by time. That could take hours to run with half a million data points. However, our computation is still very slow (even testing on only 10k points), owing to the number of comparisons we are making per point (>10) and the the distm function.

We write the following function to evaluate Local.Moving.Average.

```
Local.Moving.Average = function(frame, name, type) {
  # Sort by time
  frame$dates = as.Date(frame$Sale.Date, "%m/%d/%Y")
  frame = frame[order(as.Date(frame$dates, format="%m/%d/%Y")),]

  # Iterate rows to find homes within mile
  for (i in 1:nrow(frame)) {
    index = i - 1
    total = 0
    num_samples = 0
    prev_dist = 1000000000 # hacky - pick a really big number to find initial minimum
    closest_price = 0

    # Within 3000m
    while (index > 0 && num_samples < 10) {
      dist = distm (c(frame[i,]$lon, frame[i,]$lat), c(frame[index,]$lon, frame[index,]$lat), fun = dis

      if (which.min(c(dist, prev_dist)) == 1) {
        closest_price = frame[index, ]$Sale.Price
      }
```

```
      # Sum closest points
      if (dist < 5000) {
        total = total + frame[index, ]$Sale.Price
        num_samples = num_samples + 1
      }
      index = index - 1
      prev_dist = dist
    }

    # Pick either the closest data if nothing was close enough, or the moving average
    if (total == 0) {
      frame$Local.Moving.Average[i] = closest_price
    } else {
      frame$Local.Moving.Average[i] = total / num_samples
    }
  }
  return(frame)
}
```

We add Local.Moving.Average, and we see that is it significant and our r-squared value improves to 38%. This is much better!

```
                        Pr(>|t|)
(Intercept)             0.0634815 .
Market.Value           < 2.22e-16 ***
Taxable.Land           < 2.22e-16 ***
Taxable.Building       < 2.22e-16 ***
Exempt.Land            1.4398e-09 ***
Frontage                0.2676295
Depth                  2.0630e-05 ***
Number.of.Rooms        2.7959e-06 ***
Number.of.Bedrooms      0.0025528 **
Number.of.Bathrooms    < 2.22e-16 ***
Log.Total.Livable.Area < 2.22e-16 ***
Local.Moving.Average   < 2.22e-16 ***
```

Since Frontage is not significant, we remove it from the model. When we do so, we achieve an r-squared of 38% still.

Next, we want to try to evaluate the model using different alpha values. We try 1 (Lasso), 0.5 (ElasticNet) and 0 (Ridge) penalties. All of these methods yield lower r-squared values and higher 2013 and 2008 RMSE values, so we opt to use linear regression with engineered features in our model.

Additionally, it is worth testing different confidence values (the threshold we use to specify whether or not we ought to purchase) and seeing how these affect our RMSE values. With 5000, we get 2008 RMSE of 79055541891 and 2013 RMSE of 8854856609, which equate to being off by about $281,168 in 2008 $94,100 on a group of 20,000 homes. We can also try with a lower threshold, 2,500. The 2008 and 2013 RMSE's are the same, as they are with a threshold of 500. We opt to use the threshold of 500 (see conclusion for limiting fators).

We conclude that the best fit is the linear fit given as given in the model above.

Final Model

Our final fit has an r-squared of 38% and a 2008 error of $527,188 / 2013 error of $310,700 over 100,000 homes. This is a strong performance.

First, Market.Value represents the tax assessed value of the home, an assessment which is made every few years. While it helps us predict the value of a home, it is not actually accurate to the market value. Taxable.Land and Taxable.Building capture a similar effect, including some writeoffs.

Depth is the amount of space reserved for, for example, a sidewalk in a home's front yard. The more space reserved, the lower a home's value. This makes sense. In a broader sense, the variable probably captures some amount of geographic effect since we would may expect less expensive areas to take more of this space.

Number of rooms and bathrooms positively correlate with the value of the home, which is not surprising.

However, Number.of.Bedrooms negatively affects values. This owes to its correlation with the previous two variables; hence, it is capturing a more subtle effect explained by the number of bedrooms. We see a similarly effect with Log.Total.Livable.Area, which is also surprising. Again, it correlates with Taxable.Land, so it is capturing a subtlety of increasing the size of larger properties.

Finally, Local.Moving.Average captures the effect of nearby sold houses. As expected, the higher the value of nearby sold homes, the higher the value our model predicts.

Conclusion

There are many interesting directions in which this project can be taken moving forward. First, it would be useful to deploy the project on a platform such as Hadoop or Spark which is capable of handling the large number of computations needed to create a variable such as Local.Moving.Average. This would greatly speed up processing time, which proved to be a huge bottleneck in developing models. It would then also be possible, for example, to iteratively calculate the optimum decision threshold or test many more values of alpha, rather than using ad hoc methods to pick what we think may be the best values.

Secondly, geographic data (such as shapefiles and satellite imagery) could be integrated as new variables by using functions that manipulate the current data frame. For example, we could create a variable for school district by using the shapefiles OpenDataPhilly makes available and an R library for calculating if a latitute/longitude is within each shapefile for a categorical variable.

Finally, we could shrink the data set to find homes that had been re-sold within a short period of time and identify how a change in features had affect the home price. This could help clarify how to improve home value between purchase and sale (the second aspect of the business model).

Nonetheless, with our current model, we can produce fairly strong predictions of Philadelphia home prices, probably sufficiently good enough to make home purchases without worrying about losing signficant amounts of money (e.g., a 2008 error of $281,168 should be acceptable).

Appendix

Reference

Opendoor EPE post: https://labs.opendoor.com/residual-modeling

Opendoor system design post: https://labs.opendoor.com/2015/08/23/iterative-model-development

Code

```
hw_due_date <- "23 December 2016"

devtools::install_github("dkahle/ggmap")
devtools::install_github("hadley/ggplot2")
library(rjson)
library(ggplot2)
library(ggmap)
library(geosphere)
library(glmnet)

# Working directory set here
setwd("/Users/joefarned/Desktop/repos/real-estate")

# Ad hoc data cleaning. We take properties_dirty.csv and crate properties_clean.csv.
properties_dirty = read.csv("data/properties_dirty.csv", header=T, na.string=c("", "?"))

# Illustrations of bad data
plot(properties_dirty$Other.Building)
plot(properties_dirty$View)

# Variables to be removed at this step
variables_out = c("Owner.2", "Mailing.Care.of", "Mailing.Address.1", "Mailing.Address.2", "Mailing.Stree

# Variables to be removed at this step
variables_out = c(variables_out, "Parcel.Number", "Owner.1", "Street.Name", "Registry.Number")

plot(properties_dirty$Shape)

variables_out = c(variables_out, "Street.Code", "Suffix", "Unit", "House.Extension", "Building.Code", "

variables_out = c(variables_out, "Total.Liveable.Area", "Topography", "Number.Stories", "Year.Built", "

properties_clean = properties_dirty[, !(names(properties_dirty) %in% variables_out)]

properties_clean$Sale.Price = as.numeric(sub('$','',as.character(properties_clean$Sale.Price),fixed=TRU
properties_clean$Market.Value = as.numeric(sub('$','',as.character(properties_clean$Market.Value),fixed=
properties_clean$Taxable.Land = as.numeric(sub('$','',as.character(properties_clean$Taxable.Land),fixed=
properties_clean$Taxable.Building = as.numeric(sub('$','',as.character(properties_clean$Taxable.Buildin
properties_clean$Exempt.Land = as.numeric(sub('$','',as.character(properties_clean$Exempt.Land),fixed=T
properties_clean$Exempt.Building = as.numeric(sub('$','',as.character(properties_clean$Exempt.Building)

options(digits=15)

properties_clean$Coordinates = lapply(properties_clean$Coordinates, as.character)

arg = lapply(properties_clean$Coordinates, function(x) { strsplit(x, " ")[[1]][1] })
properties_clean$lat = as.numeric(substr(arg, 2, nchar(arg) - 1))

arg = lapply(properties_clean$Coordinates, function(x) { strsplit(x, " ")[[1]][2] })
properties_clean$lon = as.numeric(substr(arg, 1, nchar(arg) - 1))

properties_clean = properties_clean[ , !(names(properties_clean)) %in% c("Coordinates")]
```

```r
properties_clean = properties_clean[!(rowSums(is.na(properties_clean))),]

write.csv(properties_clean, "data/properties_clean.csv")

philadelphia_map = get_map(location = "philadelphia", maptype = "satellite", zoom = 11)

ggmap(philadelphia_map, extent = "device") + geom_point(aes(x = lon, y = lat), colour = "red", alpha = (

hist(properties_clean$Total.Area)
hist(properties_clean$Total.Livable.Area)

hist(log(properties_clean$Total.Area))
hist(log(properties_clean$Total.Livable.Area))

hist(log(properties_clean$Market.Value))
hist(log(properties_clean$Taxable.Land))
hist(log(properties_clean$Taxable.Building))

head(summary(as.factor(properties_clean$Zip.Code)))

head(summary(as.factor(properties_clean$Sale.Date)))

hist(properties_clean$Sale.Price)
hist(log(properties_clean$Sale.Price))

Log.Total.Area = function(frame, name, type) {
  frame$Log.Total.Area = log(frame$Total.Area)
  frame$Log.Total.Area = sapply(frame$Log.Total.Area, function(x) if (is.infinite(x)) { x = 0 } else {x
  return(frame)
}

Local.Moving.Average = function(frame, name, type) {
  # Sort by time
  frame$dates = as.Date(frame$Sale.Date, "%m/%d/%Y")
  frame = frame[order(as.Date(frame$dates, format="%m/%d/%Y")),]

  # Iterate rows to find homes within mile
  for (i in 1:nrow(frame)) {
    index = i - 1
    total = 0
    num_samples = 0
    prev_dist = 1000000000 # hacky - pick a really big number to find initial minimum
    closest_price = 0

    # Within 3000m
    while (index > 0 && num_samples < 10) {
      dist = distm (c(frame[i,]$lon, frame[i,]$lat), c(frame[index,]$lon, frame[index,]$lat), fun = dis

      if (which.min(c(dist, prev_dist)) == 1) {
        closest_price = frame[index, ]$Sale.Price
      }

      # Sum closest points
```

```
      if (dist < 5000) {
        total = total + frame[index, ]$Sale.Price
        num_samples = num_samples + 1
      }
      index = index - 1
      prev_dist = dist
    }

    # Pick either the closest data if nothing was close enough, or the moving average
    if (total == 0) {
      frame$Local.Moving.Average[i] = closest_price
    } else {
      frame$Local.Moving.Average[i] = total / num_samples
    }
  }
  return(frame)
}
                         Pr(>|t|)
(Intercept)             0.0634815 .
Market.Value           < 2.22e-16 ***
Taxable.Land           < 2.22e-16 ***
Taxable.Building       < 2.22e-16 ***
Exempt.Land            1.4398e-09 ***
Frontage                0.2676295
Depth                  2.0630e-05 ***
Number.of.Rooms        2.7959e-06 ***
Number.of.Bedrooms      0.0025528 **
Number.of.Bathrooms    < 2.22e-16 ***
Log.Total.Livable.Area < 2.22e-16 ***
Local.Moving.Average   < 2.22e-16 ***

Local.Moving.Average = function(frame, name, type) {
  # Sort by time
  frame$dates = as.Date(frame$Sale.Date, "%m/%d/%Y")
  frame = frame[order(as.Date(frame$dates, format="%m/%d/%Y")),]

  # Iterate rows to find homes within mile
  for (i in 1:nrow(frame)) {
    index = i - 1
    total = 0
    num_samples = 0
    prev_dist = 1000000000 # hacky - pick a really big number to find initial minimum
    closest_price = 0

    # Within 3000m
    while (index > 0 && num_samples < 10) {
      dist = distm (c(frame[i,]$lon, frame[i,]$lat), c(frame[index,]$lon, frame[index,]$lat), fun = dis

      if (which.min(c(dist, prev_dist)) == 1) {
        closest_price = frame[index, ]$Sale.Price
      }

      # Sum closest points
```

```
      if (dist < 5000) {
        total = total + frame[index, ]$Sale.Price
        num_samples = num_samples + 1
      }
      index = index - 1
      prev_dist = dist
    }

    # Pick either the closest data if nothing was close enough, or the moving average
    if (total == 0) {
      frame$Local.Moving.Average[i] = closest_price
    } else {
      frame$Local.Moving.Average[i] = total / num_samples
    }
  }
  return(frame)
}

Log.Market.Value = function(frame, name, type) {
  frame$Log.Market.Value = log(frame$Market.Value)
  frame$Log.Market.Value = sapply(frame$Log.Market.Value, function(x) if (is.infinite(x)) { x = 0 } else
  return(frame)
}

Log.Total.Area = function(frame, name, type) {
  frame$Log.Total.Area = log(frame$Total.Area)
  frame$Log.Total.Area = sapply(frame$Log.Total.Area, function(x) if (is.infinite(x)) { x = 0 } else {x
  return(frame)
}

Log.Total.Livable.Area = function(frame, name, type) {
  frame$Log.Total.Livable.Area = log(frame$Total.Livable.Area)
  frame$Log.Total.Livable.Area = sapply(frame$Log.Total.Livable.Area, function(x) if (is.infinite(x)) {
  return(frame)
}

Log.Taxable.Land = function(frame, name, type) {
  frame$Log.Taxable.Land = log(frame$Taxable.Land)
  sapply(frame$Log.Taxable.Land, function(x) if (is.infinite(x)) { x = 0 } else {x = x})
  return(frame)
}

Log.Taxable.Building = function(frame, name, type) {
  frame$Log.Taxable.Building = log(frame$Taxable.Building)
  frame$Log.Taxable.Building = sapply(frame$Log.Taxable.Building, function(x) if (is.infinite(x)) { x =
  return(frame)
}

Log.Sale.Price = function(frame, name, type) {
  frame$Log.Sale.Price = log(frame$Sale.Price)
  frame$Log.Sale.Price = sapply(frame$Log.Sale.Price, function(x) if (is.infinite(x)) { x = 0 } else {x
  return(frame)
}
```

```
set_variable_type = function(frame, name, type) {
  # Categorical
  if (type == "categorical") {
    frame[, (names(frame) %in% c(name))] = as.factor(frame[, (names(frame) %in% c(name))])
  }
  return(frame)
}

init = function() {
  properties_clean = read.csv("data/properties_clean.csv", header=T, na.string=c("", "?"), stringsAsFac
}

model_rmse = function(fit, epe_model, frame, start_date, end_date, confidence_threshold) {
  # Check confidence in purchase
  in_date_range = as.Date(frame$Sale.Date, "%m/%d/%Y") > start_date
  error = predict(epe_model, frame)
  should_purchase = (error < confidence_threshold) & in_date_range

  # Predict price and profit
  purchase_price = predict(fit, frame)
  rmse = sum((frame$Sale.Price[should_purchase] - purchase_price[should_purchase])^2)
  return (rmse)
}

# Main
run_experiment = function(frame, config_file, mode) {
  config = fromJSON(file=paste("config/", config_file, sep=""))

  # Use a segment of data if we are only testing
  if (mode == "test") {
    experiment_frame = frame[1:10000, ]
  } else {
    experiment_frame = frame
  }

  # Set up response
  experiment_frame = set_variable_type(experiment_frame, names(config$response), config$response[1])

  index = 0
  for (f in config$features) {
    # Add to regression formula
    if (index == 0) {
      formula = names(f)
    } else {
      formula = paste(formula, "+", names(f))
    }
    index = index + 1

    # Set up variable
    if (exists(names(f))) {
      generator_function = get(names(f))
      experiment_frame = generator_function(experiment_frame, names(f), f[[1]])
    } else {
```

```
    experiment_frame = set_variable_type(experiment_frame, names(f), f[[1]])
  }
}

# Split into training, test, validation
spec = c(train = .6, test = .2, validate = .2)
set.seed(1)
g = sample(cut(
  seq(nrow(experiment_frame)),
  nrow(experiment_frame) * cumsum(c(0,spec)),
  labels = names(spec)
))
res = split(experiment_frame, g)

experiment_frame.train = res$train
experiment_frame.test = res$test
experiment_frame.validate = res$validate

if (config$model_type == 'linear_regression') {
  print(formula)
  fit = lm(paste("Sale.Price ~", formula), experiment_frame.train)
  print(summary(fit))
} else if (config$model_type == 'elastic_net') {
  X = as.matrix(experiment_frame.train[, !(names(frame) %in% c("Sale.Price"))])
  Y = as.matrix(experiment_frame.train[, (names(frame) %in% c("Sale.Price"))])
  set.seed(1)
  cv = cv.glmnet(X, Y, alpha = config$alpha)
  pred = predict(lasso.mod,type="coefficients",s=cv$lambda.min)
  coef_names = names(pred[which(pred != 0), ])
  formula = paste(coef_names[-1], collapse = "+")
  print(formula)
  fit = lm(paste("Sale.Price ~", formula), experiment_frame.train)
  print(summary(fit))
}

# Create EPE model
experiment_frame.train$prediction_error = (predict(fit, experiment_frame.train) - experiment_frame.tra
epe_model = lm(paste("prediction_error ~", formula), experiment_frame.train)

# Calculate expected profts
rmse2008 = model_rmse(fit, epe_model, experiment_frame.validate, as.Date("2008-1-1"), as.Date("2009-1-
rmse2013 = model_rmse(fit, epe_model, experiment_frame.validate, as.Date("2013-1-1"), as.Date("2014-1-

# Save the results
config$results[1] = list(
  list("2008 rmse" = rmse2008)
)

config$results[2] = list(
  list("2013 rmse" = rmse2013)
)

config$results[3] = list(
```

```
    list("r.squared" = summary(fit)$r.squared)
  )

  config$results[4] = list(
    list("rmse" = summary(fit)$sigma)
  )

  write(toJSON(config), paste("results/", format(Sys.time(), "%b_%d_%Y_%H:%M:%S"), sep=""))
}
```