Introduction

In this project, we implemented linear regression with L2 (quadratic) regularization which learns an weight vector w to optimize Sum of Squared Error objective(SSE) using Batch Gradient Algorithm. We aim to understand the difference between linear regression without regularization and with regularization in the value of SSE through this project.

In part 0, we discuss preprocessing and simple analysis. Preprocessing helps us in utilizing data in training by making us think which feature might be useful, in which way we can transform the data to get a better result such as binarization and normalization.

In part 1, we explore the influence of learning rate. In this section, we found the overfitting problem and found that when the learning rate is big, it is easy to get explode issue. However, when the learning rate is small, it is extremely to get the answer in life time. Surprisingly, we found that most of the converged answer without regularization is overfitted.

Then we tried a lot in part 2 for the experiment with regularization. We trained the data with different learning rate and record different training and validation SSE and the number of iterations does the training need. Amazingly and surprisingly, we were excited about our observation just prove the equation. Any two number of iterations make same ratio with the ratio of their learning rate with a same regularization. Furthermore, we found that when is increasing the training SSE is also increasing monodically, while the validation SSE would increase first then drop down with a relative large . However, sometimes, a really large will also makes algorithm get exploded as our observations. In fact, with a good regularization, the converged model could behave much better on validation SSE.

In part 3, we used non-normalization data to train. In this section, we found that it is really hard to find a enough small learning rate for training without normalization. Beside, without normalization, the gradient also becomes very huge and needs more time for running the algorithm. In addition, at the end of this project, we performed training using the binarized data, which is the idea mentioned in preprocessing part 0.b for exploring more details about the idea in Part 3. b (Extra). We binarized some features like date and zipcode. Binarizing date doesn’t improve the result a lot, however, binarizing zipcode decrease our validation SSE over 30% off. Finally we used the model which are binarized in both date and zipcode feature for predicting the result of test file.

Part 0. Preprocessing and simple analysis

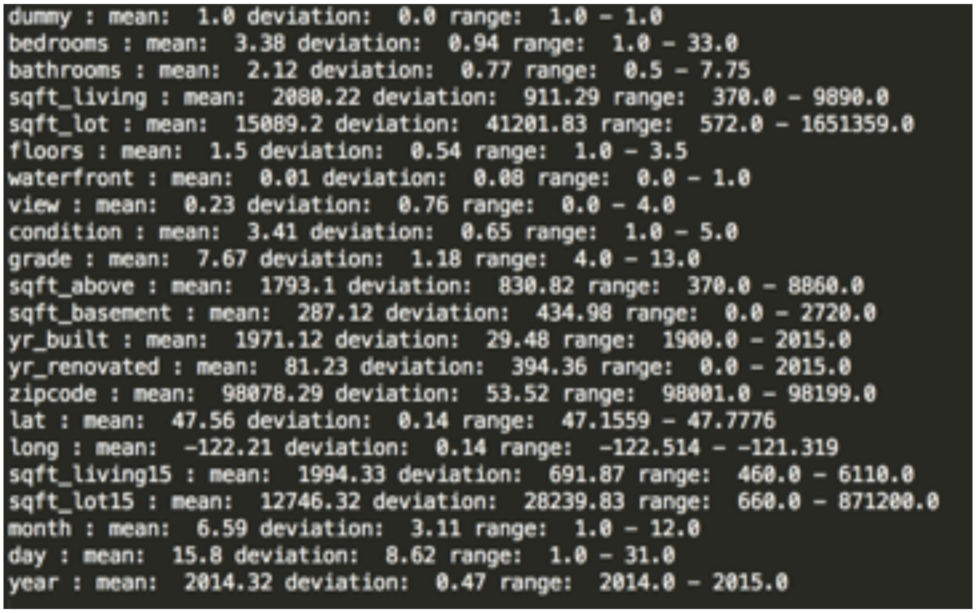
a. Remove the ID feature. Why do you think it is a bad idea to use it in learning?

: Because the ID feature almost doesn’t have any relationships with the house price like using the feature of today’s weather to predict the result of tossing a normal coin. Extremely speaking, if we just keep and only keep this feature for predicting a toss, we also could get a non-zero weight on this feature, which means this feature will cause some influence on our result. Since we have already known the background knowledge of there is no or very rare connections between the ID feature and the house price, just like the connection between weather and toss a coin, so the influence on our result must be a bad influence. So we have to remove this feature for a better result. On the other hand, we also could treat it as a overfitting problem. Since we have one more useless feature to the result, while we have to adjust our model to fit on this useless feature. As a result, our new model must be more overfit than the removed-model. From this perspective, the result also can be improved.

b. Split the date feature into three separate numerical features

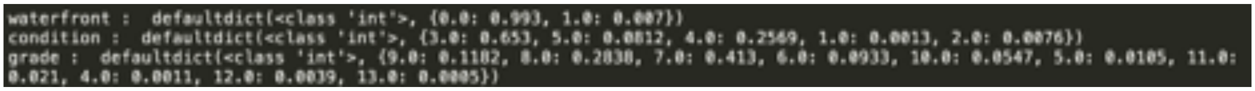
: Yes. Binarizing these separated features would be one way. After normalizing these date feature, there is still a inner linear relationship within the data which means that December have more influence than the January (12 > 1, and when we calculate w*x, the result in December could be increased inappropriately.) While, instead of separating and normalizing it, we could increase the number of features and binarize these data. Like separate date to 100 (approximate # of years) + 12 (# of month) + 31 (# of date) = 143 new features. If the date is “Oct 7th, 2018”, then the “Oct”, “7th”, and “2018” should be set to 1 and the rest field should be set to 0.

c. Build a table that reports the statistics for each feature



```
dummy : mean:  1.0 deviation:  0.0 range:  1.0 - 1.0
bedrooms : mean:  3.38 deviation:  0.94 range:  1.0 - 33.0
bathrooms : mean:  2.12 deviation:  0.77 range:  0.5 - 7.75
sqft_living : mean:  2080.22 deviation:  911.29 range:  370.0 - 9890.0
sqft_lot : mean:  15089.2 deviation:  41201.83 range:  572.0 - 1651359.0
floors : mean:  1.5 deviation:  0.54 range:  1.0 - 3.5
waterfront : mean:  0.01 deviation:  0.08 range:  0.0 - 1.0
view : mean:  0.23 deviation:  0.76 range:  0.0 - 4.0
condition : mean:  3.41 deviation:  0.65 range:  1.0 - 5.0
grade : mean:  7.67 deviation:  1.18 range:  4.0 - 13.0
sqft_above : mean:  1793.1 deviation:  830.82 range:  370.0 - 8860.0
sqft_basement : mean:  287.12 deviation:  434.98 range:  0.0 - 2720.0
yr_built : mean:  1971.12 deviation:  29.48 range:  1900.0 - 2015.0
yr_renovated : mean:  81.23 deviation:  394.36 range:  0.0 - 2015.0
zipcode : mean:  98078.29 deviation:  53.52 range:  98001.0 - 98199.0
lat : mean:  47.56 deviation:  0.14 range:  47.1559 - 47.7776
long : mean:  -122.21 deviation:  0.14 range:  -122.514 - -121.319
sqft_living15 : mean:  1994.33 deviation:  691.87 range:  460.0 - 6110.0
sqft_lot15 : mean:  12746.32 deviation:  28239.83 range:  660.0 - 871200.0
month : mean:  6.59 deviation:  3.11 range:  1.0 - 12.0
day : mean:  15.8 deviation:  8.62 range:  1.0 - 31.0
year : mean:  2014.32 deviation:  0.47 range:  2014.0 - 2015.0
```

Figure 1.



```
waterfront :  defaultdict(<class 'int'>, {0.0: 0.993, 1.0: 0.007})
condition :  defaultdict(<class 'int'>, {3.0: 0.653, 5.0: 0.0812, 4.0: 0.2569, 1.0: 0.0013, 2.0: 0.0076})
grade :  defaultdict(<class 'int'>, {9.0: 0.1182, 8.0: 0.2838, 7.0: 0.413, 6.0: 0.0933, 10.0: 0.0547, 5.0: 0.0105, 11.0:
0.021, 4.0: 0.0011, 12.0: 0.0039, 13.0: 0.0005})
```

Figure 2.

	mean	standard deviation	range
dummy	1.0	0	[1.0,1.0]
bedrooms	3.38	0.94	[1.0,33.0]
bathrooms	2.12	0.77	[0.5,7.75]
sqft_living	2080.22	911.29	[370.0,9890.0]
sqft_lot	15089.2	41201.83	[572.0,1651359.0]
floors	1.5	0.54	[1.0,3.5]
waterernt	0.01	0.08	[0.0,1.0]
view	0.23	0.76	[0.0,4.0]
condition	3.41	0.65	[1.0,5.0]
grade	7.67	1.18	[4.0,13.0]
sqft_above	1793.1	830.82	[370.0,8860.0]
sqft_basement	287.12	434.98	[0.0,2720.0]
yr_built	1971.12	29.48	[1900.0,2015.0]
yr_renovated	81.23	394.36	[0.0,2015.0]
zipcode	98078.29	53.52	[98001.0,98199.0]
lat	47.56	0.14	[47.1559,47.7776]
long	-122.21	0.14	[-122.514,-121.319]
sqft_living15	1994.33	691.87	[460.0,6110.0]
sqft_lot15	12746.32	28239.83	[660.0,871200.0]
month	6.59	3.11	[1.0,12.0]
day	15.8	8.62	[1.0,31.0]
year	2014.32	0.47	[2014.0,2015.0]

Table 1.

d. Which set of features are expected to be useful?
: We think the feature of the feature of "sqft_above" should be the most useful feature in predicting the result. Based on our common knowledge, the price of house should be based on the equation: house price = total square * the quality of the house (unit price of sqft). Thus the newer could effect the quality of the house. However, the date at this domain doesn't have a rough deviation (all of house built in 2014-2015.) Beside, some features like the condition, zipcode, waterfront, grade, floors, lat, long just have a small

deviation. Intuitively, I think those features would cause small influence on the result. Thus, the total square of the house is the most useful feature for predicting a house.

e. Normalization

: We looked up the internet about how to normalize the data, then we found four ways for normalization.

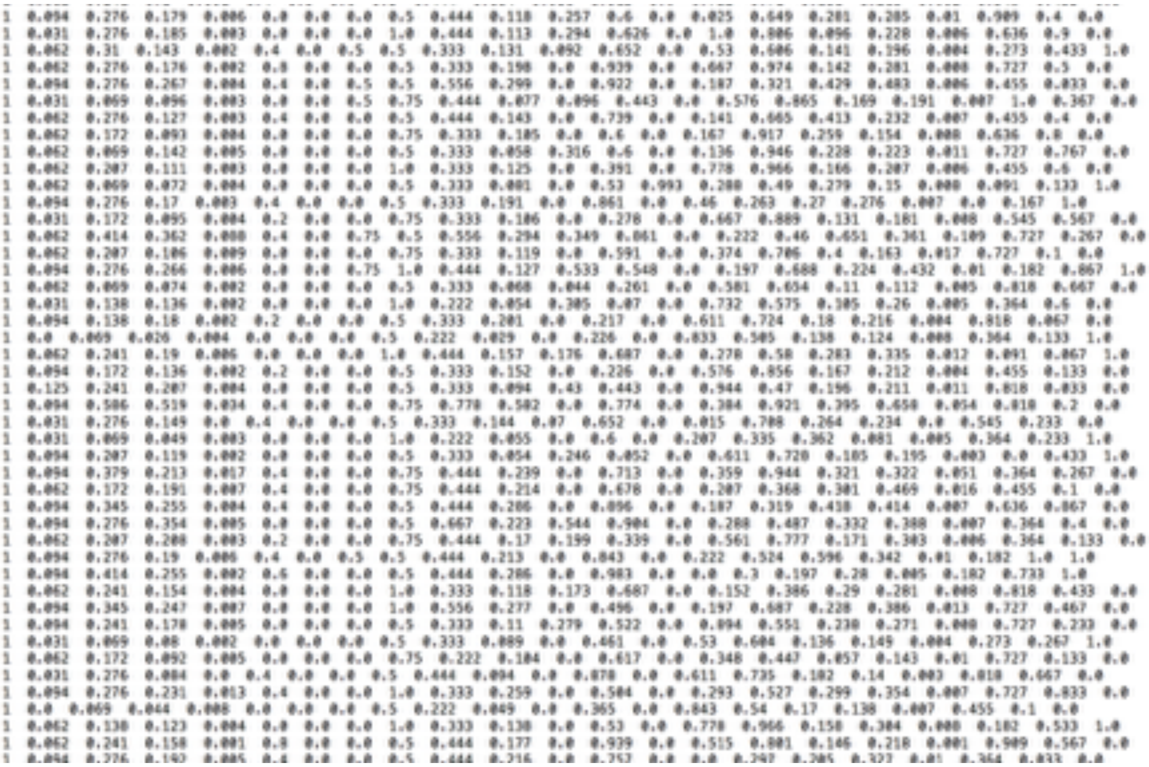
(1) $\boldsymbol{a}_1$ / sqrt(sum($\boldsymbol{a}_1^2$ +…+ $\boldsymbol{a}_n^2$) … an / sqrt(sum($\boldsymbol{a}_1^2$ +…+$\boldsymbol{a}_n^2$)

(2) $\boldsymbol{a}_1$ / max($\boldsymbol{a}_1$… $\boldsymbol{a}_n$) … an / max($\boldsymbol{a}_1$…$\boldsymbol{a}_n$)

(3) $\boldsymbol{a}$ = ($\boldsymbol{a}$ - min) / (max - min)

(4) z-score normalization: a = a - mean / standard deviation.

We will use the last one normalization method (4). If we have time, we will implement the second method and compare their different influence. Figure 2 shows a part of normalized data.



```
1 0.094 0.276 0.179 0.006 0.0 0.0 0.0 0.5 0.444 0.118 0.257 0.6 0.0 0.025 0.649 0.281 0.285 0.01 0.909 0.4 0.0
1 0.031 0.276 0.185 0.003 0.0 0.0 0.0 1.0 0.444 0.113 0.294 0.626 0.0 1.0 0.006 0.096 0.228 0.006 0.636 0.9 0.0
1 0.062 0.31 0.143 0.002 0.4 0.0 0.5 0.5 0.333 0.131 0.092 0.652 0.0 0.53 0.606 0.141 0.196 0.004 0.273 0.433 1.0
1 0.062 0.276 0.176 0.002 0.0 0.0 0.0 0.5 0.333 0.198 0.0 0.939 0.0 0.667 0.974 0.142 0.281 0.008 0.727 0.5 0.0
1 0.094 0.276 0.267 0.004 0.4 0.0 0.5 0.5 0.556 0.299 0.0 0.922 0.0 0.187 0.321 0.429 0.483 0.006 0.455 0.033 0.0
1 0.031 0.069 0.096 0.003 0.0 0.0 0.5 0.75 0.444 0.077 0.096 0.443 0.0 0.576 0.865 0.169 0.191 0.007 1.0 0.367 0.0
1 0.062 0.276 0.127 0.003 0.4 0.0 0.0 0.5 0.444 0.143 0.0 0.739 0.0 0.141 0.665 0.413 0.232 0.007 0.455 0.4 0.0
1 0.062 0.172 0.093 0.004 0.0 0.0 0.0 0.75 0.333 0.105 0.0 0.6 0.0 0.167 0.917 0.259 0.154 0.008 0.636 0.8 0.0
1 0.062 0.069 0.142 0.005 0.0 0.0 0.0 0.5 0.333 0.058 0.316 0.6 0.0 0.136 0.946 0.228 0.223 0.011 0.727 0.767 0.0
1 0.062 0.207 0.111 0.003 0.0 0.0 0.0 1.0 0.333 0.125 0.0 0.391 0.0 0.778 0.966 0.166 0.207 0.006 0.455 0.6 0.0
1 0.062 0.069 0.072 0.004 0.0 0.0 0.0 0.5 0.333 0.001 0.0 0.53 0.993 0.288 0.49 0.279 0.15 0.000 0.091 0.133 1.0
1 0.094 0.276 0.17 0.003 0.4 0.0 0.0 0.5 0.333 0.191 0.0 0.861 0.0 0.46 0.263 0.27 0.276 0.007 0.0 0.167 1.0
1 0.031 0.172 0.095 0.004 0.2 0.0 0.0 0.75 0.333 0.106 0.0 0.278 0.0 0.667 0.889 0.131 0.181 0.008 0.545 0.567 0.0
1 0.062 0.414 0.362 0.008 0.4 0.0 0.75 0.5 0.556 0.294 0.349 0.061 0.0 0.222 0.46 0.651 0.361 0.109 0.727 0.267 0.0
1 0.062 0.207 0.106 0.009 0.0 0.0 0.0 0.75 0.333 0.119 0.0 0.591 0.0 0.374 0.706 0.4 0.163 0.017 0.727 0.1 0.0
1 0.094 0.276 0.266 0.006 0.0 0.0 0.75 1.0 0.444 0.127 0.533 0.548 0.0 0.197 0.688 0.224 0.432 0.01 0.182 0.867 1.0
1 0.062 0.069 0.074 0.002 0.0 0.0 0.0 0.5 0.333 0.068 0.044 0.261 0.0 0.581 0.654 0.11 0.112 0.005 0.818 0.667 0.0
1 0.031 0.138 0.136 0.002 0.0 0.0 0.0 1.0 0.222 0.054 0.305 0.07 0.0 0.732 0.575 0.105 0.26 0.005 0.364 0.6 0.0
1 0.094 0.138 0.18 0.002 0.2 0.0 0.0 0.5 0.333 0.201 0.0 0.217 0.0 0.611 0.724 0.18 0.216 0.004 0.818 0.067 0.0
1 0.0 0.069 0.026 0.004 0.0 0.0 0.0 0.5 0.222 0.029 0.0 0.226 0.0 0.833 0.505 0.138 0.124 0.000 0.364 0.133 1.0
1 0.062 0.241 0.19 0.006 0.0 0.0 0.0 1.0 0.444 0.157 0.176 0.687 0.0 0.278 0.58 0.283 0.335 0.012 0.091 0.067 1.0
1 0.094 0.172 0.136 0.002 0.2 0.0 0.0 0.5 0.333 0.152 0.0 0.226 0.0 0.576 0.856 0.167 0.212 0.004 0.455 0.133 0.0
1 0.125 0.241 0.207 0.004 0.0 0.0 0.0 0.5 0.333 0.094 0.43 0.443 0.0 0.944 0.47 0.196 0.211 0.011 0.818 0.033 0.0
1 0.094 0.506 0.519 0.034 0.4 0.0 0.0 0.75 0.778 0.502 0.0 0.774 0.0 0.304 0.921 0.395 0.658 0.054 0.818 0.2 0.0
1 0.031 0.276 0.149 0.0 0.4 0.0 0.0 0.5 0.333 0.144 0.07 0.652 0.0 0.015 0.708 0.264 0.234 0.0 0.545 0.233 0.0
1 0.031 0.069 0.049 0.003 0.0 0.0 0.0 1.0 0.222 0.055 0.0 0.6 0.0 0.207 0.335 0.362 0.081 0.005 0.364 0.233 1.0
1 0.094 0.207 0.119 0.002 0.0 0.0 0.0 0.5 0.333 0.054 0.246 0.052 0.0 0.611 0.720 0.105 0.195 0.003 0.0 0.433 1.0
1 0.094 0.379 0.213 0.017 0.4 0.0 0.0 0.75 0.444 0.239 0.0 0.713 0.0 0.359 0.944 0.321 0.322 0.051 0.364 0.267 0.0
1 0.062 0.172 0.191 0.007 0.4 0.0 0.0 0.75 0.444 0.214 0.0 0.678 0.0 0.207 0.368 0.301 0.469 0.016 0.455 0.1 0.0
1 0.094 0.345 0.255 0.004 0.4 0.0 0.0 0.5 0.444 0.206 0.0 0.096 0.0 0.187 0.319 0.418 0.414 0.007 0.636 0.067 0.0
1 0.094 0.276 0.354 0.005 0.0 0.0 0.0 0.5 0.667 0.223 0.544 0.904 0.0 0.288 0.487 0.332 0.388 0.007 0.364 0.4 0.0
1 0.062 0.207 0.208 0.003 0.2 0.0 0.0 0.75 0.444 0.17 0.199 0.339 0.0 0.561 0.777 0.171 0.303 0.006 0.364 0.133 0.0
1 0.094 0.276 0.19 0.006 0.4 0.0 0.5 0.5 0.444 0.213 0.0 0.843 0.0 0.222 0.524 0.596 0.342 0.01 0.182 1.0 1.0
1 0.094 0.414 0.255 0.002 0.6 0.0 0.0 0.5 0.444 0.206 0.0 0.983 0.0 0.0 0.3 0.197 0.28 0.005 0.182 0.733 1.0
1 0.062 0.241 0.154 0.004 0.0 0.0 0.0 1.0 0.333 0.118 0.173 0.687 0.0 0.152 0.386 0.29 0.281 0.008 0.818 0.433 0.0
1 0.094 0.345 0.247 0.007 0.0 0.0 0.0 1.0 0.556 0.277 0.0 0.496 0.0 0.197 0.687 0.228 0.386 0.013 0.727 0.467 0.0
1 0.094 0.241 0.178 0.005 0.0 0.0 0.0 0.5 0.333 0.11 0.279 0.522 0.0 0.094 0.551 0.238 0.271 0.000 0.727 0.233 0.0
1 0.031 0.069 0.08 0.002 0.0 0.0 0.0 0.5 0.333 0.089 0.0 0.461 0.0 0.53 0.604 0.136 0.149 0.004 0.273 0.267 1.0
1 0.062 0.172 0.092 0.005 0.0 0.0 0.0 0.75 0.222 0.104 0.0 0.617 0.0 0.348 0.447 0.057 0.143 0.01 0.727 0.133 0.0
1 0.031 0.276 0.084 0.0 0.4 0.0 0.0 0.5 0.444 0.094 0.0 0.878 0.0 0.611 0.735 0.182 0.14 0.003 0.818 0.667 0.0
1 0.094 0.276 0.231 0.013 0.4 0.0 0.0 1.0 0.333 0.259 0.0 0.504 0.0 0.293 0.527 0.299 0.354 0.007 0.727 0.833 0.0
1 0.0 0.069 0.044 0.008 0.0 0.0 0.0 0.5 0.222 0.049 0.0 0.365 0.0 0.843 0.54 0.17 0.138 0.007 0.455 0.1 0.0
1 0.062 0.138 0.123 0.004 0.0 0.0 0.0 1.0 0.333 0.138 0.0 0.53 0.0 0.778 0.966 0.158 0.304 0.000 0.182 0.533 1.0
1 0.062 0.241 0.158 0.001 0.0 0.0 0.0 0.5 0.444 0.177 0.0 0.939 0.0 0.515 0.801 0.146 0.218 0.001 0.909 0.567 0.0
1 0.094 0.276 0.192 0.005 0.4 0.0 0.0 0.5 0.444 0.216 0.0 0.752 0.0 0.0 0.292 0.205 0.327 0.01 0.364 0.033 0.0
```

Figure 3.

Part 1. Explore different learning rate for batch gradient descent

a. Observation of different learning rates

: 10^{-5} is the best learning rate in my observation. More specifically, 5 * 10^{-5} is even better and faster than 10^{-5}.

10^{0},10^{-1},10^{-2},10^{-3},10^{-4} and any learning rate greater than 5.8 * 10^{-5} will make the gradient decent explode.

x-rot is the number of iterations, y-rot is the value of SSE with each learning rate:

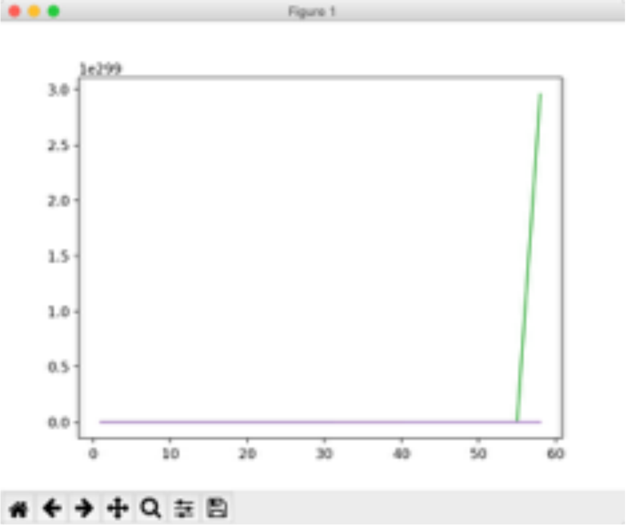


Figure 4.

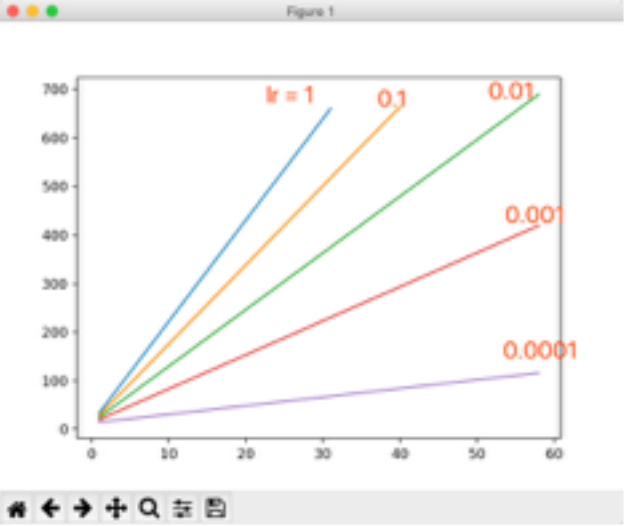


Figure 5.

In Figure 4., due to the rapid growth of the result of SSE with the growth of iterations, most of data just become a line. So we take a log for shrinking the gap among the data.

In Figure 5., we can see that the bigger learning rate it is, the more rapid growth it is. Beside, if the growth is bigger than 700, the SSE value just become "inf" and stop growing anymore. The growth of learning rate at 0.0001 is the smallest learning rate is also grow slowest. The situation above is only taking about the explode situation.

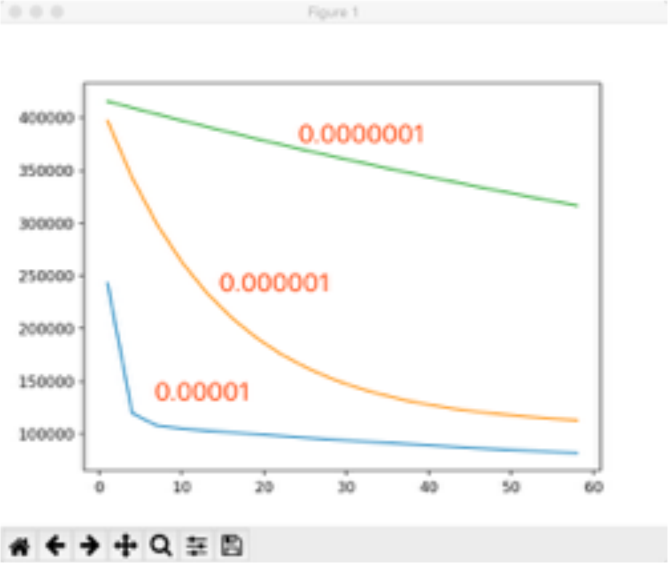


Figure 6.

In Figure 6., we can see that the tendency of these three learning rate is convergence. The bigger learning rate always brings a faster convergence speed. However, there is another issue. When the number of iterations is growing up, the speed of convergence is decreasing. Thus, we should better pick up an as big as possible learning rate for decreasing the running time. Therefore, we can conclude that the biggest learning rate cannot make a convergence. But as long as the learning rate could make a convergence, we should better use a bigger learning rate for speeding up.

b. SSE on the training data and validation data with iterations

: For setting up epsilon at 0.5 with a learning rate 0.000057, the training time is 916s with 25734 iterations. When learning rate is 0.00001, it takes more than 5 hours and didn't get a result. Thus our solution is assigning w at the beginning instead of just assigning 0 to w.

Strict unequal equation: $|\Delta E(w)|$ (interpret as getting norm of a vector) $< \epsilon$

Learning Rate	0.000057	0.00001	0.000001	0.0000001
Time	916s	5672s	-	-
Iterations	25734	146684	-	-
SSE on train	38991.5180114	38991.5180414	-	-
SSE on dev	55026.301127	55026.2647565	-	-

Table 2.

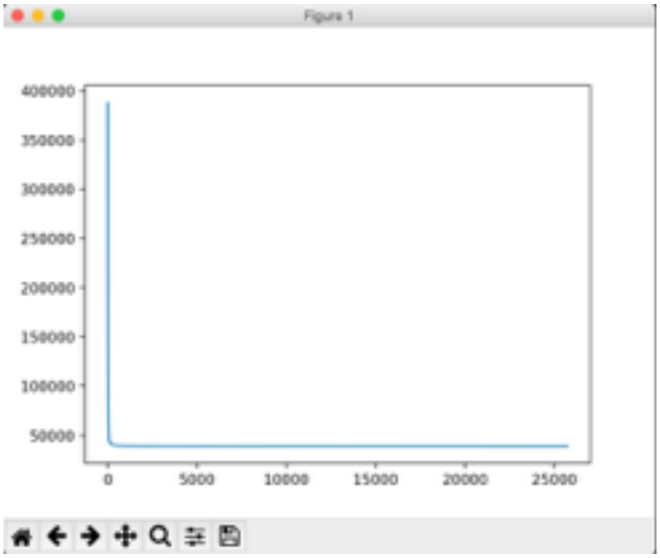


Figure 7.

Figure 7. shows that the SSE of training set with learning rate 0.000057 which may be the fastest learning rate that I chose. However, we found that in this figure, after a few iterations, the algorithm just keep going, while the SSE didn’t change a lot. Beside, with using this strict equation, it took really a long time for reaching the condition. (It took 1.5h with learning rate = 0.00001. We tried a long time with smaller learning rate, but it didn’t come out answer in 2 hours, so we just skip trying that.) However, there are too many iterations just improve a little bit for convergence. For learning rate = 0.000057, we just tried first 500 iterations for zoom-in the figure. Then;

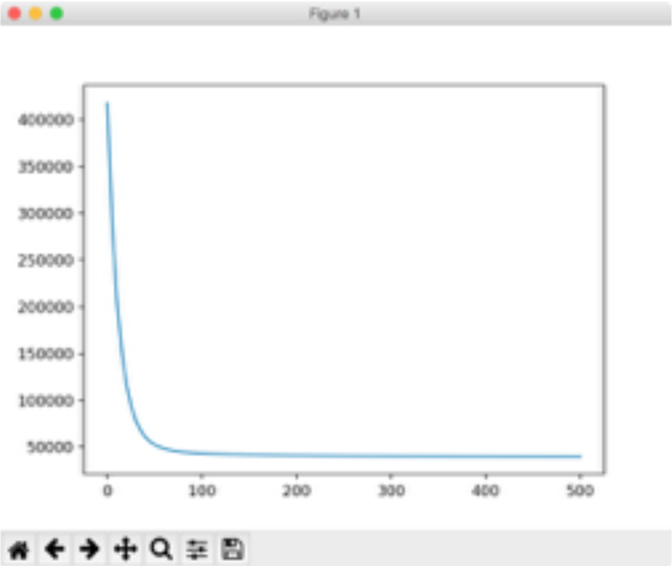


Figure 8.

Iterations	500	25000
SSE on train	39651.03976671272	38991.5180114
SSE on dev	23080.3020957	55026.301127

Table 3.

In above data set, we found out there might be an overfitting problem. (less errors on training data with well-trained model but more errors on dev data.) Hence, instead of setting an unknown loop stop point, we set up a fixed 25000 iterations, which costs 15 min per training.

learning rate = 0.000057, x-rot is # of iterations, y-rot is SSE, blue is on training data, yellow is on dev data. Figure 9. shows total SSE on each data set. Figure 10. gives average SSE on each data set.

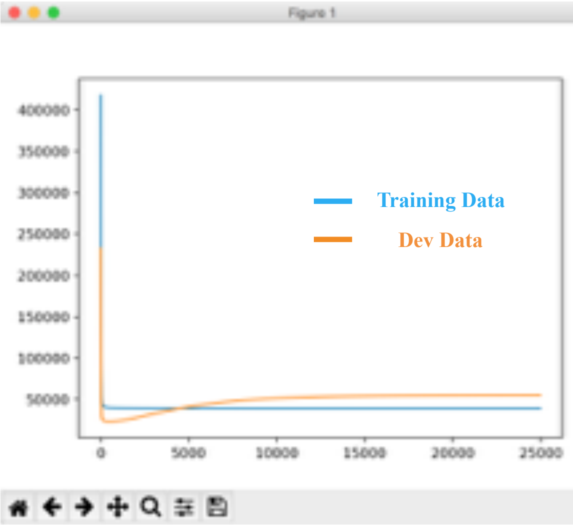


Figure 9.

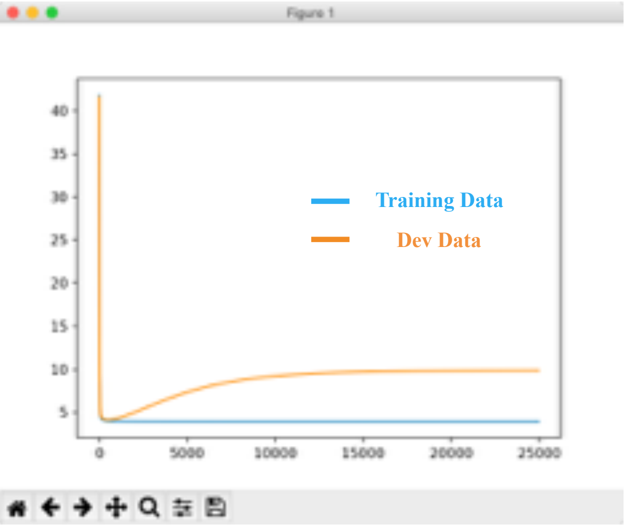


Figure 10.

So at the first drop of SSE on dev data, SSE turns to increase a lot. When we take an average SSE, the gap becomes more obvious. We think that it could prove the existence of overfitting problem. Beside, more iterations and large learning rate doesn't guarantee a good model. For showing a more clear result, we take a log function for shrinking the huge gap and magnifying the small gap:

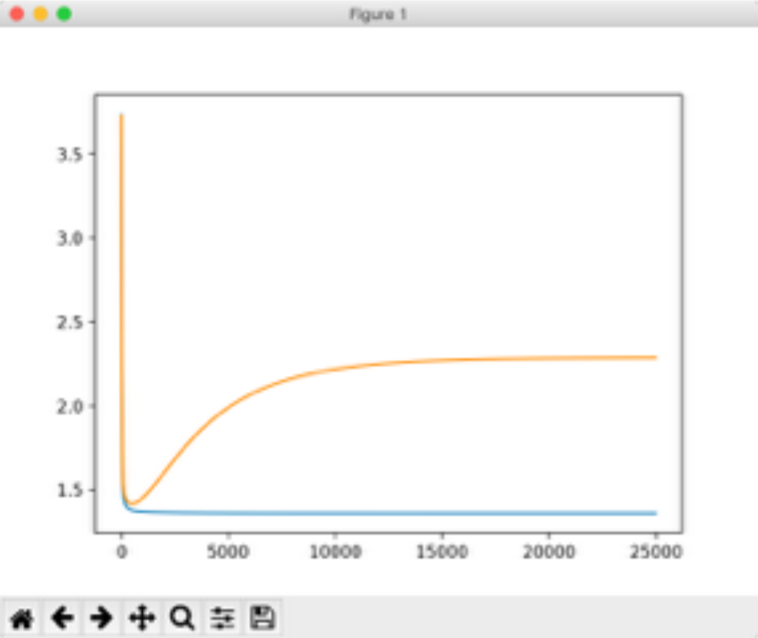


Figure 11. Learning rate = 0.000057, Y-rot = average log(SSE)

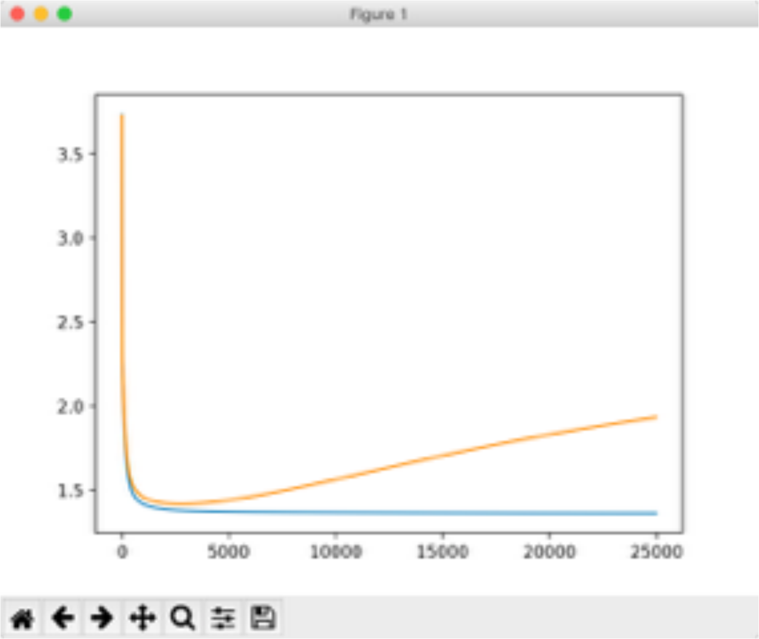


Figure 12. Learning rate = 0.00001, Y-rot = average log(SSE)

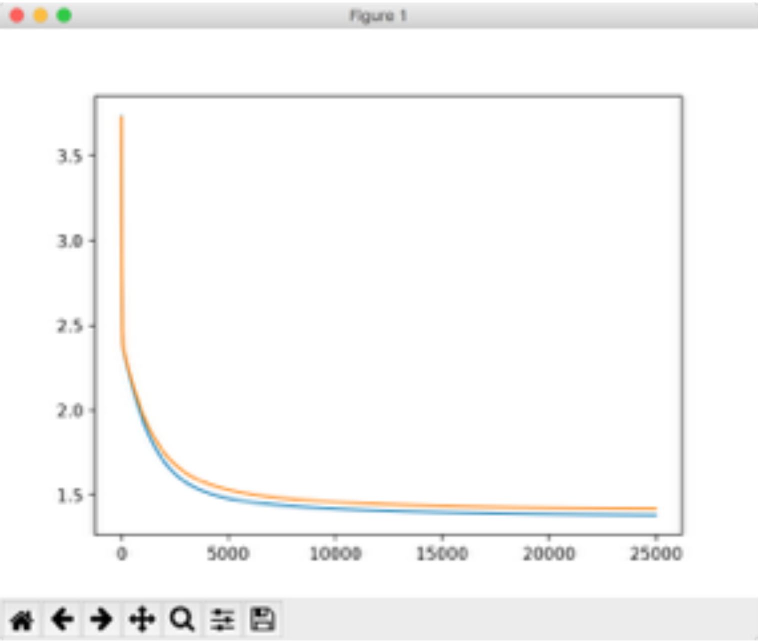


Figure 13. Learning rate = 0.000001, Y-rot = average log(SSE)

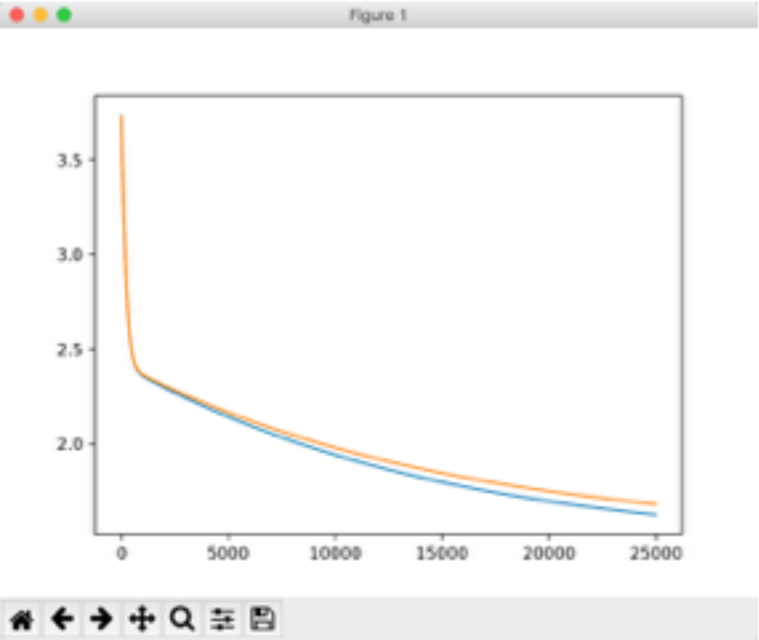


Figure 14. Learning rate = 0.0000001, Y-rot = average log(SSE)

From above observations, we found that some facts.

1. The average SSE of dev data are always higher than the average SSE of training data. It's hard to surpass, otherwise it should be an overfitting problem on the dev data set.

2. A lower learning rate needs more iterations and more time for convergence. From the two data we got, when learning rate a is 5.7 times to learning rate b. The iterations and the cost time does learning rate a needs only rough 1/5.7 to what the learning rate b needs. (a = 5.7* 10^{-5}, b = 1*10^{-5}, while # of iterations ratio is $a/b = 146684/25734 = 5.7$). So it takes around 1.5h for getting the result with learning rate b = 1*10^{-5}, we assume that it may take 15h with learning rate 1*10^{-6} and 150h with learning rate 10^{-7}.

3. The total SSE on training data is decreasing monotonically, while total SSE on dev data is decreasing first and then turn to increase a little bit with a fixed model learned from training set. We believe that is the case of overfitting problem, with the model more fitting on the training data, it starts to be not fitting on dev data.

4. From 2, we know that a lower learning rate will decrease the speed of training, while it could extend the period of decreasing of SSE on dev data. When we want to train a model, we don't want a model which only could fit on current data. So our goal should be getting the lowest SSE of testing on dev data. However, the lowest SSE of dev data with a model which is trained with a higher learning rate cannot be found easily. In our examples, when learning rate is 5.7*10^{-5}, the lowest SSE value only lasts a short time. Thus, the higher learning rate isn't always mean a good thing. Sometimes, it cannot let us to find the lowest SSE value on dev data. (We almost missed it, because at the beginning, we only tried the model after convergence which may have already been the overfitted model.)

5. When learning rate is equal to 10^{-6} and 10^{-7}, there is not an increase of SSE on dev data. However, we assume that there isn't an increase of SSE on dev data so far, because there isn't enough iterations for them to get more fit on the training data. So we assume may be after 25000 iterations (10^{-5} change its tendency after around 2500 iterations), the SSE on dev data with learning rate 10^{-6} will turn to increase and SSE on dev data with learning rate 10^{-7} will change its tendency after 250000 iterations.

6. The lower learning rate cannot guarantee a lower SSE on dev data set.

   LR = 5.7 * 10^{-5} -> minimum SSE 23078.986037
   LR = 1.0 * 10^{-5} -> minimum SSE 23080.3943119

c. Learned weights for each feature
: When learning rate is equal to 10^{-5}, the number of iterations for convergence is over 140 thousand. In this feature weight, the most absolute biggest feature is the "number of bedrooms".

It is similar to our pre-analysis because according to our approximate prediction equation: size of the house * quality of the house. Although we didn't think the most biggest feature is the "# of bedrooms", however, the most top 4 features are "# of bedroom", "sqft_living", "sqft_above" which are representing the size of the house. Then "grade" represents the quality of the house. Thus, this fit on our rough prediction.

Features	weight
dummy	-2.02634726
bedrooms	-9.1811135
bathrooms	3.33191851
sqft_living	7.26277625
sqft_lot	2.31366786
floors	0.11770157
waterernt	4.71107643
view	2.32284388
condition	1.25636608
grade	8.8744548
sqft_above	7.72517044
sqft_basement	1.3069606
yr_built	-2.9363435
yr_renovated	0.28851062
zipcode	-1.02065174
lat	3.76772534
long	-2.67437416
sqft_living15	1.33430012
sqft_lot15	-2.91738238
month	0.18228678
day	-0.17278705
year	0.38104872

Table 4.

Beside, the feature weight of month, day, year are also very small and just like what we predicted. However, the lower feature weight somehow represent the relationship between the feature and prediction label. It's really hard to assert that the contribution of deviation, mean and range to the feature weight. (Like "year" feature has a low deviation while "day" feature has a relative higher deviation, but both of them don't have a larger weight.) Thus, we guess that the lower feature weight only relate to the connection to the prediction label (in this domain, it should be the relationship between feature and predicted price.)

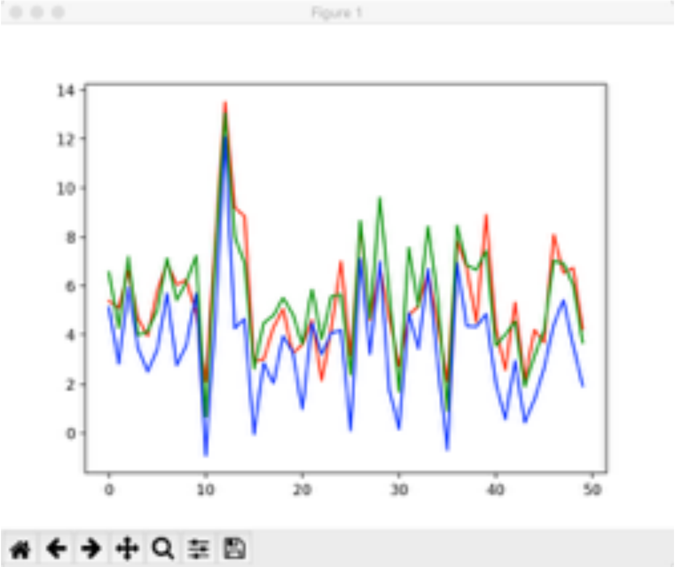
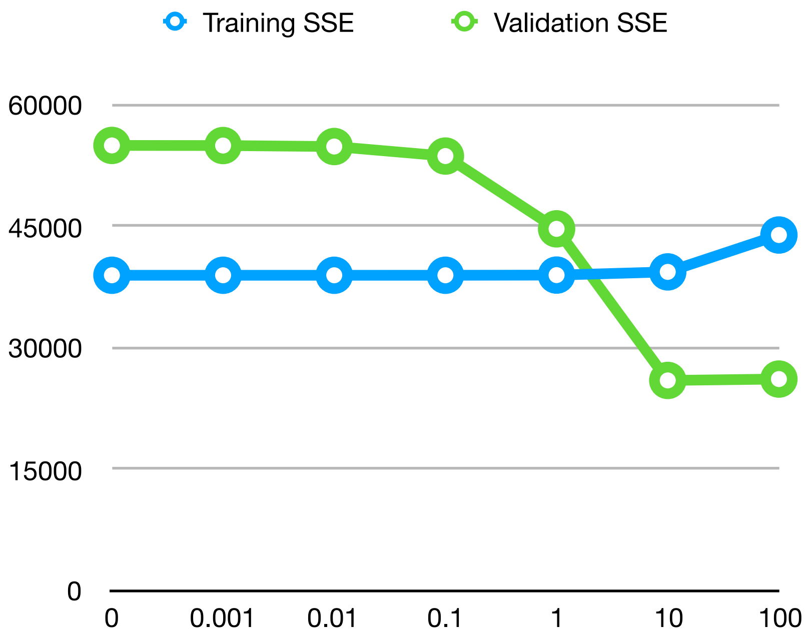


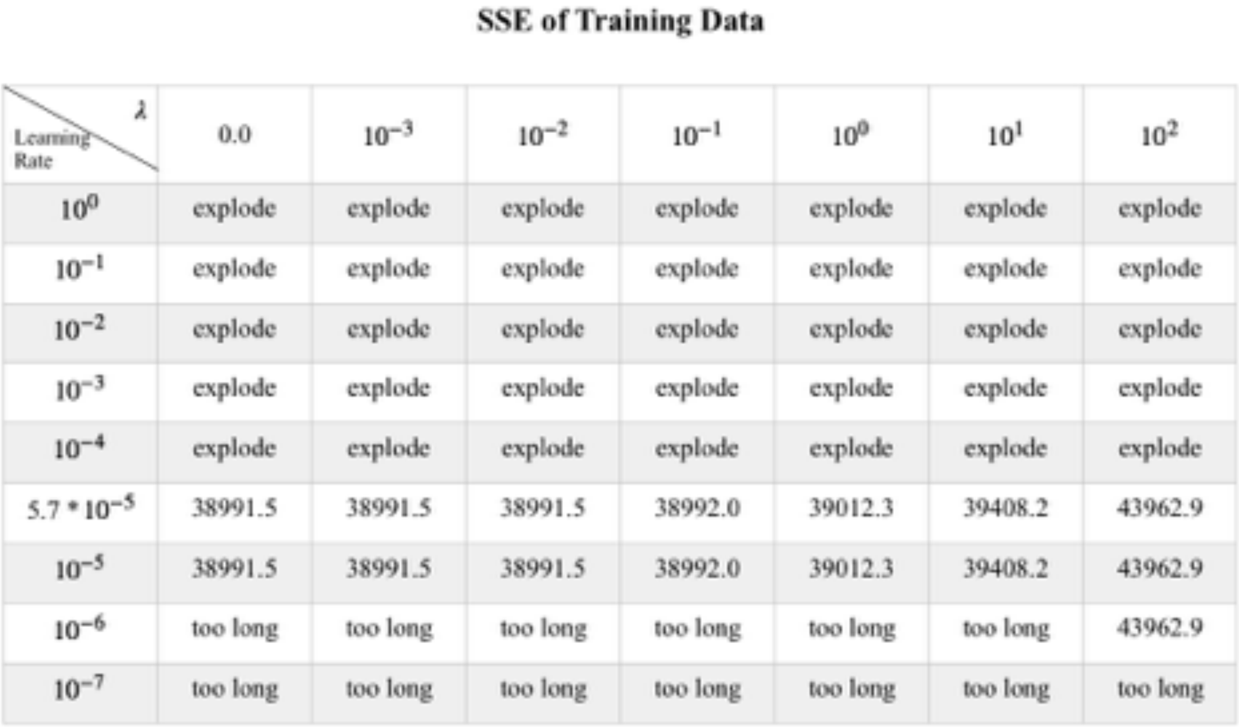
Figure 15.

Figure 15. shows that the first 50 predicted result of predicting in dev data. Red line is real data, green line is the model of 484 iterations model with learning rate 0.000057, blue line is the model of fully convergence 25734 iterations overfitting model with learning rate 0.000057.

Part 2. Experiments with different λ values



Graph 1.

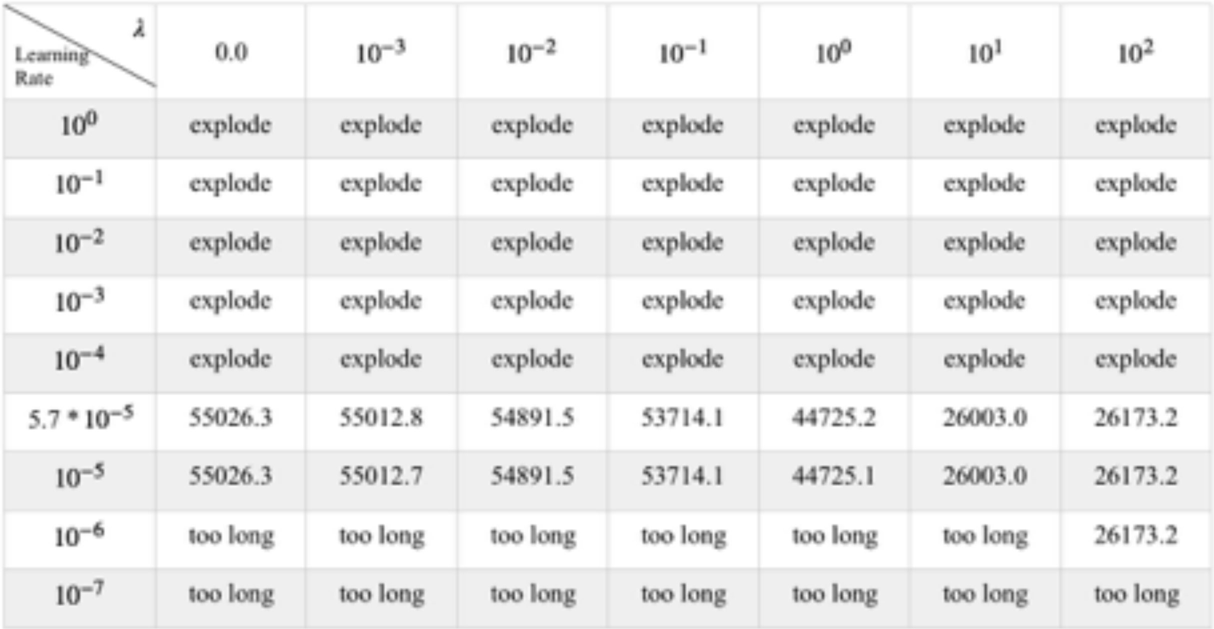


SSE of Training Data

Learning Rate \ λ	0.0	10^{-3}	10^{-2}	10^{-1}	10^{0}	10^{1}	10^{2}
10^{0}	explode	explode	explode	explode	explode	explode	explode
10^{-1}	explode	explode	explode	explode	explode	explode	explode
10^{-2}	explode	explode	explode	explode	explode	explode	explode
10^{-3}	explode	explode	explode	explode	explode	explode	explode
10^{-4}	explode	explode	explode	explode	explode	explode	explode
$5.7 * 10^{-5}$	38991.5	38991.5	38991.5	38992.0	39012.3	39408.2	43962.9
10^{-5}	38991.5	38991.5	38991.5	38992.0	39012.3	39408.2	43962.9
10^{-6}	too long	too long	too long	too long	too long	too long	43962.9
10^{-7}	too long	too long	too long	too long	too long	too long	too long

Table 5.

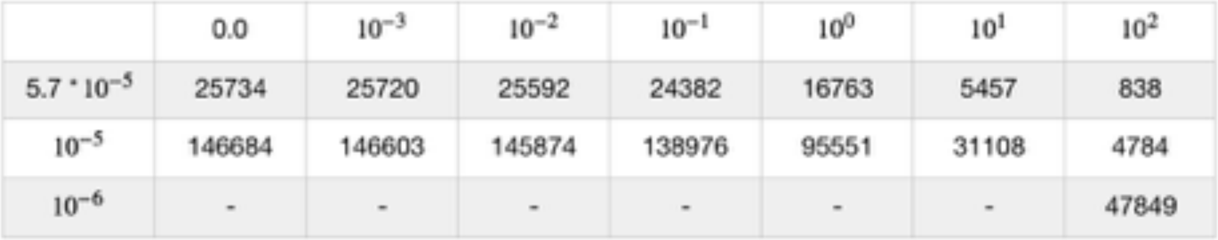
SSE of Dev Data



Learning Rate \ λ	0.0	10^{-3}	10^{-2}	10^{-1}	10^{0}	10^{1}	10^{2}
10^{0}	explode	explode	explode	explode	explode	explode	explode
10^{-1}	explode	explode	explode	explode	explode	explode	explode
10^{-2}	explode	explode	explode	explode	explode	explode	explode
10^{-3}	explode	explode	explode	explode	explode	explode	explode
10^{-4}	explode	explode	explode	explode	explode	explode	explode
$5.7 * 10^{-5}$	55026.3	55012.8	54891.5	53714.1	44725.2	26003.0	26173.2
10^{-5}	55026.3	55012.7	54891.5	53714.1	44725.1	26003.0	26173.2
10^{-6}	too long	too long	too long	too long	too long	too long	26173.2
10^{-7}	too long	too long	too long	too long	too long	too long	too long

Table 6.

Iterations for Convergence



	0.0	10^{-3}	10^{-2}	10^{-1}	10^{0}	10^{1}	10^{2}
$5.7 * 10^{-5}$	25734	25720	25592	24382	16763	5457	838
10^{-5}	146684	146603	145874	138976	95551	31108	4784
10^{-6}	-	-	-	-	-	-	47849

Table 7.

a. Trend from the training SSE with change of λ value
: When λ is going up, the training SSE is increasing monotonically.

b. rend from the validation SSE with change of λ value
: When λ is going up, the validation SSE is decreasing at the beginning and increasing at the end with a large λ.

c. Explanation of observed behaviors
: In part I, without any regularization, due to our batch gradient algorithm, the training SSE should decrease monodically for sure. So the convergence point is also the point where the lowest training SSE we can get. However, after adding a larger and larger λ, we are not only

speed up the training speed, but also decrease the accuracy on training data set. More specifically, the loss function is

$$\frac{1}{2}(\Sigma_{i=1}^{N}(x_i * w - y_i)^2 + \lambda||w||^2)$$

then the gradient should be;

$$\Sigma_{i=1}^{N}(x_i * w - y_i)x_i + \lambda w$$

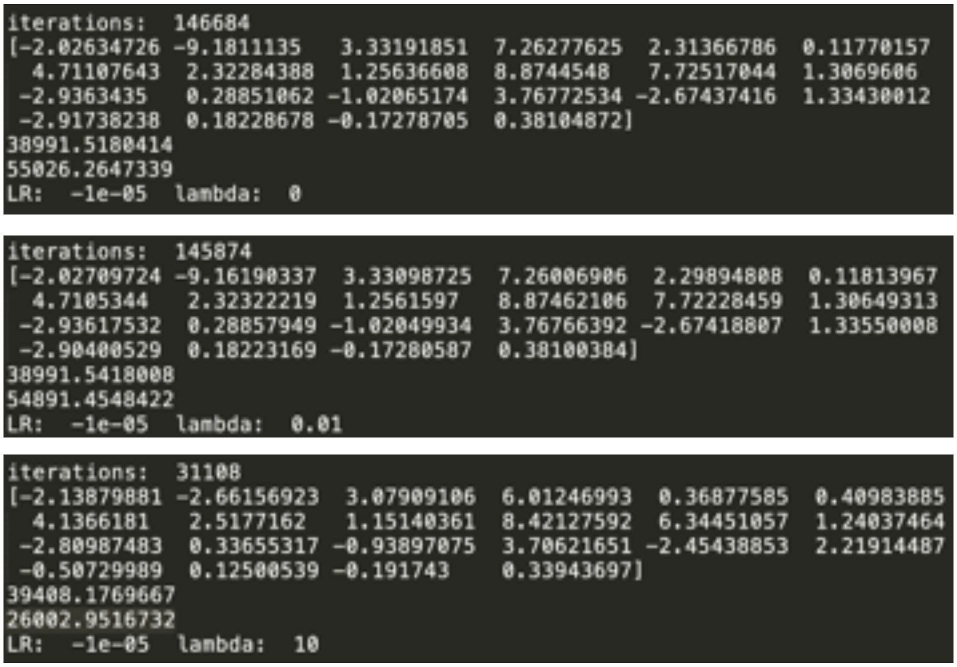
Then, we could use the result of learning rate by gradient for updating the value of w. However, when the learning rate is a fixed value and λ is increasing, the model w is using larger and larger itself for updating itself. Thus, the term we used in class of "regularization" is for controlling how fast the w update itself. We list the # of iterations with 2 different learning rate. Thus, with a larger λ, the speed of training is faster for sure (because some iterations will be skipped when w update itself, then it only needs fewer iterations for convergence). At the beginning of this explanation, we also mentioned about that the λ=0 training could get the best result on training SSE for sure. When the w skipped some its iterations in λ=0, it must get a relative worse training SSE (It kind of like, when you are going down straight of the mountain with same speed, somehow, the lowest point of the mountain just increase a little bit, suddenly we just don't need too many time (iterations) for reaching the lowest point.) Thus, when λ is going up, the training SSE is increasing monodically due to the regularization.

Beside, when λ is going up, the validation SSE is decreasing at the beginning and increasing at the end with a large λ. Because the result we get in part 1.b is showing that there is some overfitting issue within the 25000 iterations training with learning rate both 0.000057 and 0.00001. The theory best model (get the lowest point on both training SSE and validation SSE) should only take around 480 iterations within a learning rate of 0.000057. So when the lambda is bigger, while the regularization is rough, the model just skipped a lot of useless updates and just try to keep and save the best model to the end of training. However, when the λ is small, the influence of λw is too small to the result, so the regularization is not strong and the result is trend to the result without any regularization. Our the third table shows the number of iterations does the model needs for convergence. Furthermore, when the λ is too big (we tried 100000 as a λ), this large-λ model just skipped almost updates, even include those best models (it only take 146 iterations when λ=100000, while the best model appears around 480th iterations). Therefore, we conclude this phenomenon in the issue of overfitting problem. When λ is not big enough, the regularization is stronger and strong, then overfitting problem become less and less. As a result, the validation SSE become better and better. However, when λ is too big, the regularization is too strong, then the model just too sparse and also cannot get a good validation SSE.

Sometimes, when λ is really too big, the lower learning rate also got exploded.

d. What features get turned off for λ= 10, 10^{-2} and 0 ?

:



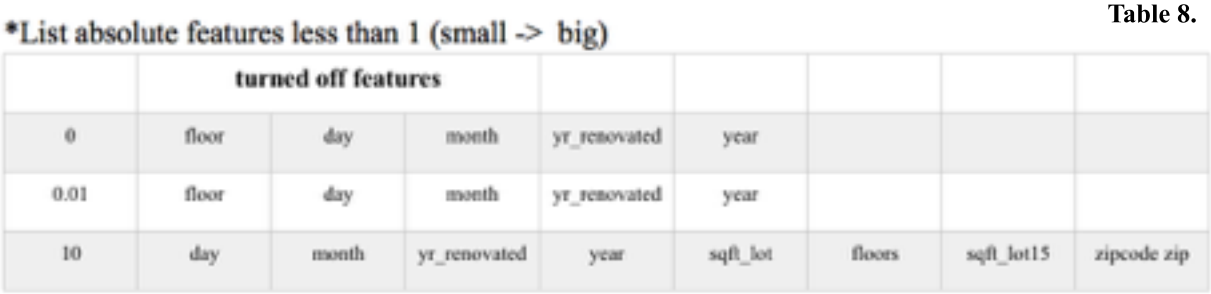
```
iterations:  146684
[-2.02634726 -9.1811135   3.33191851  7.26277625  2.31366786  0.11770157
  4.71107643  2.32284388  1.25636608  8.8744548   7.72517044  1.3069606
 -2.9363435   0.28851062 -1.02065174  3.76772534 -2.67437416  1.33430012
 -2.91738238  0.18228678 -0.17278705  0.38104872]
38991.5180414
55026.2647339
LR:  -1e-05  lambda:  0
```

```
iterations:  145874
[-2.02709724 -9.16190337  3.33098725  7.26006906  2.29894808  0.11813967
  4.7105344   2.32322219  1.2561597   8.87462106  7.72228459  1.30649313
 -2.93617532  0.28857949 -1.02049934  3.76766392 -2.67418807  1.33550008
 -2.90400529  0.18223169 -0.17280587  0.38100384]
38991.5418008
54891.4548422
LR:  -1e-05  lambda:  0.01
```

```
iterations:  31108
[-2.13879881 -2.66156923  3.07909106  6.01246993  0.36877585  0.40983885
  4.1366181   2.5177162   1.15140361  8.42127592  6.34451057  1.24037464
 -2.80987483  0.33655317 -0.93897075  3.70621651 -2.45438853  2.21914487
 -0.50729989  0.12500539 -0.191743    0.33943697]
39408.1769667
26002.9516732
LR:  -1e-05  lambda:  10
```

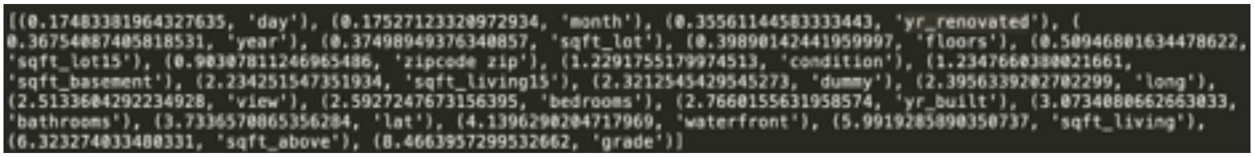
Figure 16.

Table 8.



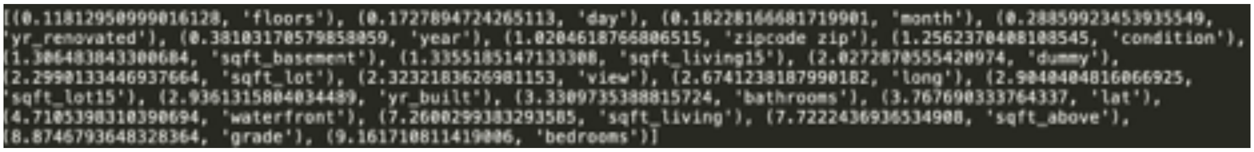
*List absolute features less than 1 (small -> big)

	turned off features							
0	floor	day	month	yr_renovated	year			
0.01	floor	day	month	yr_renovated	year			
10	day	month	yr_renovated	year	sqft_lot	floors	sqft_lot15	zipcode zip



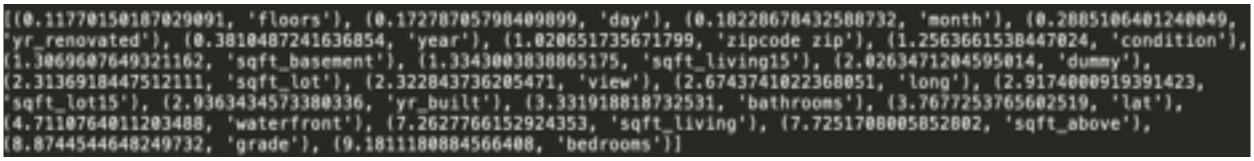
```
[(0.17483381964327635, 'day'), (0.17527123320972934, 'month'), (0.35561144583333443, 'yr_renovated'), (
0.36754087405818531, 'year'), (0.37498949376340857, 'sqft_lot'), (0.39890142441959997, 'floors'), (0.50946801634478622,
'sqft_lot15'), (0.90307811246965486, 'zipcode zip'), (1.2291755179974513, 'condition'), (1.2347660380021661,
'sqft_basement'), (2.234251547351934, 'sqft_living15'), (2.3212545429545273, 'dummy'), (2.3956339202702299, 'long'),
(2.5133604292234928, 'view'), (2.5927247673156395, 'bedrooms'), (2.7660155631958574, 'yr_built'), (3.0734080662663033,
'bathrooms'), (3.7336570065356284, 'lat'), (4.1396290204717969, 'waterfront'), (5.9919285090350737, 'sqft_living'),
(6.323274033480331, 'sqft_above'), (8.4663957299532662, 'grade')]
```

Figure 17. $\lambda = 10$



```
[(0.11812950999016128, 'floors'), (0.1727894724265113, 'day'), (0.18228166681719901, 'month'), (0.28859923453935549,
'yr_renovated'), (0.38103170579858059, 'year'), (1.0204618766806515, 'zipcode zip'), (1.2562370408108545, 'condition'),
(1.306483843300684, 'sqft_basement'), (1.3355185147133308, 'sqft_living15'), (2.0272870555420974, 'dummy'),
(2.2990133446937664, 'sqft_lot'), (2.3232183626981153, 'view'), (2.6741238187990182, 'long'), (2.9040404816066925,
'sqft_lot15'), (2.9361315804034489, 'yr_built'), (3.3309735388815724, 'bathrooms'), (3.767690333764337, 'lat'),
(4.7105398310390694, 'waterfront'), (7.2600299383293585, 'sqft_living'), (7.7222436936534908, 'sqft_above'),
(8.8746793648328364, 'grade'), (9.161710811419006, 'bedrooms')]
```

Figure 18. $\lambda = 0.01$



```
[(0.11770150187029091, 'floors'), (0.17278705798409899, 'day'), (0.18228678432588732, 'month'), (0.2885106401240049,
'yr_renovated'), (0.3810487241636854, 'year'), (1.020651735671799, 'zipcode zip'), (1.2563661538447024, 'condition'),
(1.3069607649321162, 'sqft_basement'), (1.3343003838865175, 'sqft_living15'), (2.0263471204595014, 'dummy'),
(2.3136918447512111, 'sqft_lot'), (2.322843736205471, 'view'), (2.6743741022368051, 'long'), (2.9174000919391423,
'sqft_lot15'), (2.9363434573380336, 'yr_built'), (3.331918818732531, 'bathrooms'), (3.7677253765602519, 'lat'),
(4.7110764011203488, 'waterfront'), (7.2627766152924353, 'sqft_living'), (7.7251708005852802, 'sqft_above'),
(8.8744544648249732, 'grade'), (9.1811180884566408, 'bedrooms')]
```

Figure 19. $\lambda = 0$

When the λ change from 0 to 0.01, there is not a big change. However, when the λ from 0.1 to 10, there are much more features whose absolute weight is less than 1. Beside, when $\lambda = 10$, some features we discussed in 2.c are becoming smaller. Like the weight of # of bedrooms, from 9 to 3. Which is more fitting on our intuition: the square of above should have more connections with the price rather than the # of bedrooms. As a result, the validation SSE of $\lambda = 10$ is much better than the validation SSE without regularization. Therefore, after adding a moderate regularization as a punishment should improve the convergence result of the algorithm.

e. Other observations

: We also have some other observations:

First, the learning rate which get explode in part 1 also cannot get converge with adding the regularization. However, some learning rate (like 0.000057) with a relative bigger λ will also cause an explode.

Second, after adding the regularization, the ratio between # of iteration and learning rate is still same. Like it takes around 5.7 times iterations compare LR=0.000057 and 0.00001.

Third, after adding same regularization, both training SSE and validation SSE are almost same with different learning rate.

Part 3. Training with non-normalized data and binarization

a. Observations from non-normalized data

:

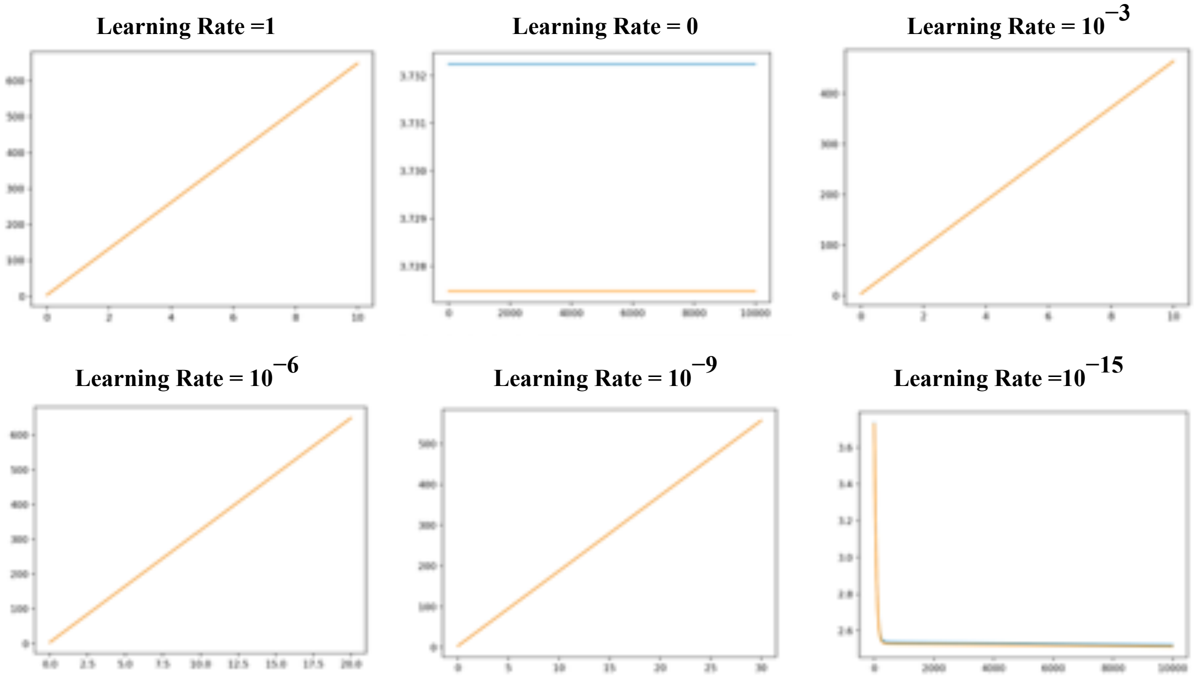
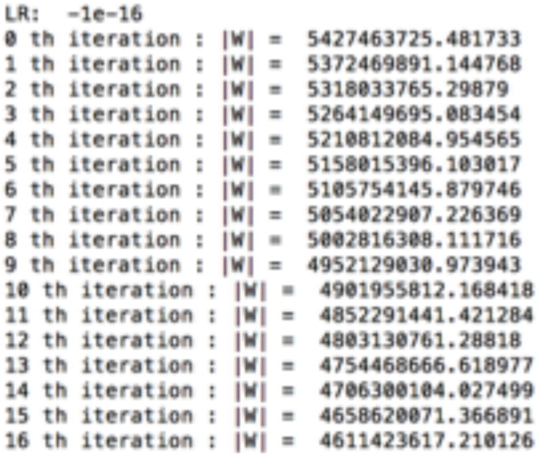


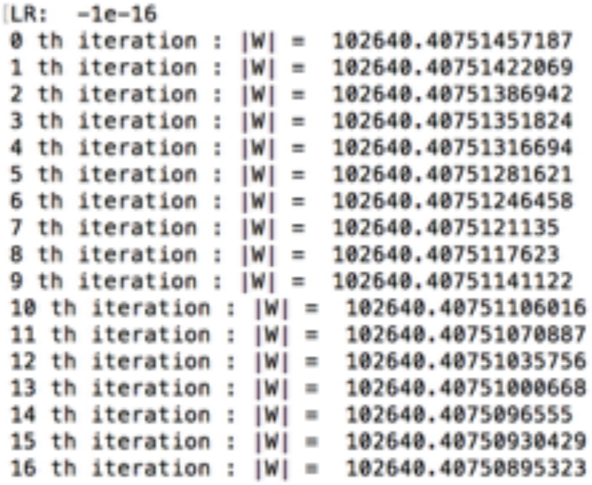
Figure 20.

Figure 20. shows the training SSE(Blue line) and validation SSE(Orange line) as a function of the number of iterations(x-rot). According to the graph, we can see that Learning Rate = 1, 10^{-3}, 10^{-6}, and 10^{-9} drives SSE both in training data and validation data explode. When learning rate is zero, SSE does not change since our program does not learn. However, when the learning rate is 10^{-15}, which is very small, we can finally prevent the the gradient descent from exploding. As a proof, we can see the last graph in Figure 20., the only graph showing SSE decreasing with increase of iterations.



```
LR:  -1e-16
0 th iteration : |W| =  5427463725.481733
1 th iteration : |W| =  5372469891.144768
2 th iteration : |W| =  5318033765.29879
3 th iteration : |W| =  5264149695.083454
4 th iteration : |W| =  5210812084.954565
5 th iteration : |W| =  5158015396.103017
6 th iteration : |W| =  5105754145.879746
7 th iteration : |W| =  5054022907.226369
8 th iteration : |W| =  5002816308.111716
9 th iteration : |W| =  4952129030.973943
10 th iteration : |W| =  4901955812.168418
11 th iteration : |W| =  4852291441.421284
12 th iteration : |W| =  4803130761.28818
13 th iteration : |W| =  4754468666.618977
14 th iteration : |W| =  4706300104.027499
15 th iteration : |W| =  4658620071.366891
16 th iteration : |W| =  4611423617.210126
```

Figure 21.

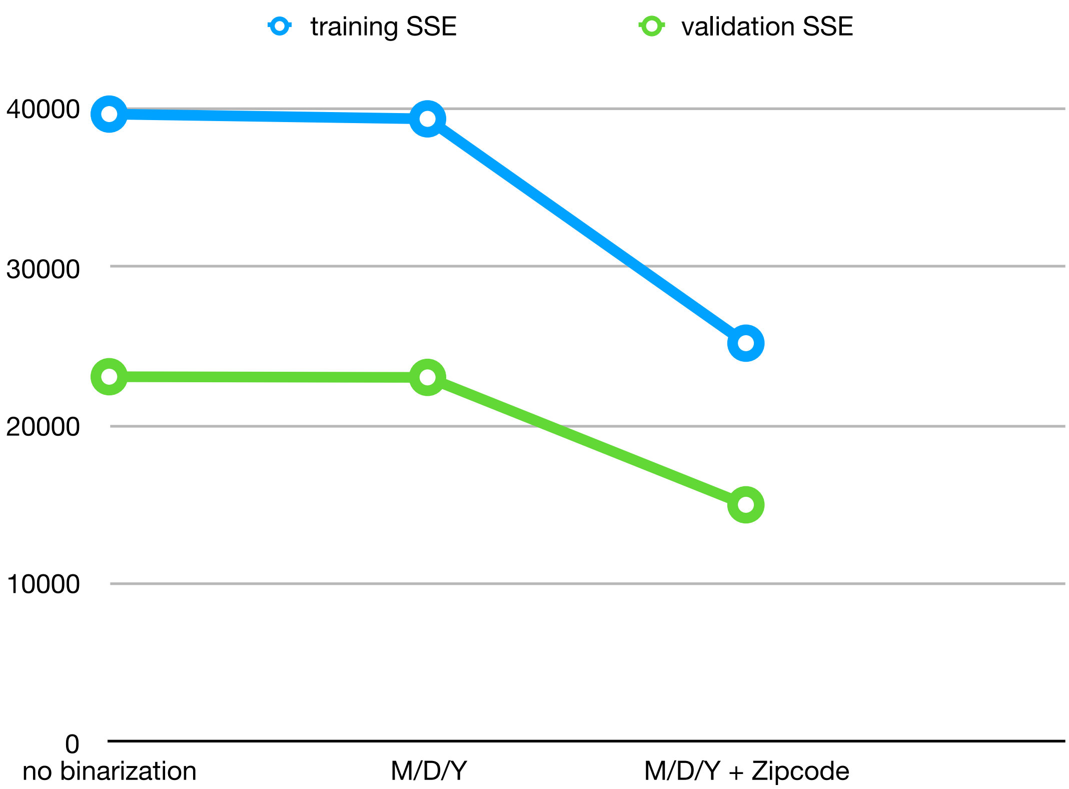


```
LR:  -1e-16
0 th iteration : |W| =  102640.40751457187
1 th iteration : |W| =  102640.40751422069
2 th iteration : |W| =  102640.40751386942
3 th iteration : |W| =  102640.40751351824
4 th iteration : |W| =  102640.40751316694
5 th iteration : |W| =  102640.40751281621
6 th iteration : |W| =  102640.40751246458
7 th iteration : |W| =  102640.4075121135
8 th iteration : |W| =  102640.4075117623
9 th iteration : |W| =  102640.40751141122
10 th iteration : |W| =  102640.40751106016
11 th iteration : |W| =  102640.40751070887
12 th iteration : |W| =  102640.40751035756
13 th iteration : |W| =  102640.40751000668
14 th iteration : |W| =  102640.4075096555
15 th iteration : |W| =  102640.40750930429
16 th iteration : |W| =  102640.40750895323
```

Figure 22.

Figure 21. shows the change of length of the gradient |∇E(w)| with the increase of integration on non-normalized data while Figure 22. presents it on normalized data. Those two figures informs that training the non-normalized data puts us in huge scale computing problem. At the 0-th iteration, the value of |∇E(w)| of non-normalized data is 5000 times bigger than the one of normalized data. This causes overflow in computing the norm of the vector, and requires smaller learning rate. For example, when we used normalized data, the best choice of our learning rate is 0.000057. However, in non-normalized data, when the learning rate reaches to 10^{-15}, it finally started to work properly. 10^{-9} is not small enough for non-normalized data. We could find that normalization causes the gradient shrink more with larger learning rate than the non-normalization in the same fixed iterations. We realized that all those facts of normalization makes the training the data easier.

b. Observations from binarized data (Extra)
:

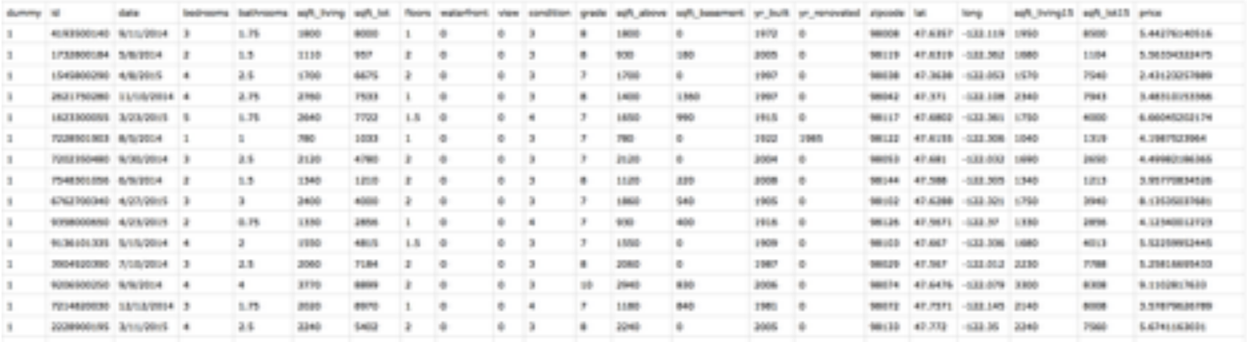


Graph 2.

We mentioned about in part 0.b, we would like to try some binarization for the feature which doesn't have a linear positive correlation. So in this extra part, we binarize the date features into month*12 + day*31 +year*2 features. Then we found out that this binarization could increase in the result indeed. First, our criteria of the result is based on both the lowest validation SSE and the smaller gap between average training SSE and validation SSE. Then, after binarization, the training SSE from 39674.46 to 39373.04 while validation SSE from 23078.986 to 23036.46, which may not be a great improvement, but it decrease the total SSE of both training and validation data.

Furthermore, we didn't use the category name given by professor. As our assumption, we believe that the zipcode feature should be more independent to each other zipcode. Thus, we binarize month, dat, year, zipcode in total. Then, the best training SSE drops dramatically, from 39373.04 to 25198.19 in training SSE and 23036.46 to 14985.41.

Then we check the weight of each dimension of the w, we found out that the most important weight becomes a "zipcode" (we didn't mark which zipcode is the most important.) We assume that the house price based on zipcode between Corvallis and Silicon Valley must be different, then after binarizing the zipcode, this differences could be shown from the training process. Also, the date information also the relative small weight in w. Which may explain that why just binarizing date information only improve a small SSE. Thus, we tested using this model to test the PA1_test.csv file. Figure 23. shows the part of the results.



dummy	id	date	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade	sqft_above	sqft_basement	yr_built	yr_renovated	zipcode	lat	long	sqft_living15	sqft_lot15	price
1	4193500140	9/11/2014	3	1.75	1800	8000	1	0	0	3	8	1800	0	1972	0	98008	47.6357	-122.119	1950	8500	5.44276140516
1	1732800184	5/8/2014	2	1.5	1110	957	2	0	0	3	8	930	180	2005	0	98119	47.6319	-122.362	1680	1104	5.56334332475
1	1545800290	4/8/2015	4	2.5	1700	6675	2	0	0	3	7	1700	0	1997	0	98038	47.3638	-122.053	1570	7540	2.43123257889
1	2621750280	11/10/2014	4	2.75	2760	7533	1	0	0	3	8	1400	1360	1997	0	98042	47.371	-122.108	2340	7943	3.48510153566
1	1623300055	3/23/2015	5	1.75	2640	7722	1.5	0	0	4	7	1650	990	1915	0	98117	47.6802	-122.361	1750	4000	6.66045202174
1	7228501903	8/5/2014	1	1	780	1033	1	0	0	3	7	780	0	1922	1985	98122	47.6155	-122.306	1040	1319	4.1987523964
1	7202350480	9/30/2014	3	2.5	2120	4780	2	0	0	3	7	2120	0	2004	0	98053	47.681	-122.032	1690	2650	4.49982186365
1	7548301056	6/9/2014	2	1.5	1340	1210	2	0	0	3	8	1120	220	2008	0	98144	47.588	-122.305	1340	1213	3.95770834526
1	6762700340	4/27/2015	3	3	2400	4000	2	0	0	3	7	1860	540	1905	0	98102	47.6288	-122.321	1750	3940	8.13535037681
1	9358000650	4/23/2015	2	0.75	1330	2856	1	0	0	4	7	930	400	1916	0	98126	47.5671	-122.37	1330	2856	4.12540012723
1	9136101335	5/15/2014	4	2	1550	4815	1.5	0	0	3	7	1550	0	1909	0	98103	47.667	-122.336	1680	4013	5.52259932445
1	3904920390	7/10/2014	3	2.5	2060	7184	2	0	0	3	8	2060	0	1987	0	98029	47.567	-122.012	2230	7788	5.25816605433
1	9206500250	9/9/2014	4	4	3770	8899	2	0	0	3	10	2940	830	2006	0	98074	47.6476	-122.079	3300	8308	9.1102817610
1	7214820030	12/12/2014	3	1.75	2020	8970	1	0	0	4	7	1180	840	1981	0	98072	47.7571	-122.145	2140	8008	3.57879626789
1	2228900195	3/11/2015	4	2.5	2240	5402	2	0	0	3	8	2240	0	2005	0	98133	47.772	-122.35	2240	7560	5.6741163031

Figure 23.