Assignment 1 : Studying the Linear Regression Model

Suyash Damle
15CS10057

Language used: Python (2.7)

Libraries/Modules used:

- pandas.dataframe: for R-like functionality in reading data and initial tabulation. Only used in the initial phase – during data reading and to separate the train and test tuples.
- pandas.read_csv: for reading the data from the csv file
- numpy : for efficient numerical array/matrix storage and operations. The library supports very quick matrix multiplication, slicing and other operations.
- math.sqrt
- matplotlib.pyplot: for plotting operations

Some Specifications about the implementation:

- As there are 4 features in the dataset, the number of features/learned parameters are 5 everywhere (including $x_0 = 1$)

- Z-score normalization has been used for all the features **and the target values as well** in all the codes:
  Z-norm(x) = (x-mean(A)) / std. dev(A), where x is an instance of the attribute A

- The initial vector for the coefficients is **initialized randomly at each call of the grad_dec()** function. I have observed slight difference in the final model based on the initial values picked up. Hence, the **RMSE reported in different runs of the code may vary slightly** because of the initial assignment.

- In part (c), while taking product of features, the newly-created features are **normalized after the multiplication operation**, so that the data is always normalized exactly before being passed to learning function.

- Wherever the number of iterations are not to be passed to the *grad_desc()* function, the stopping condition is taken to be the convergence of the cost value. The value that is checked is the ratio of cost reduction to the original cost, ie: (old_cost – new_cost) / old_cost

- In the part(c), while taking quadratic combinations all feature combinations upto 2nd order terms are taken : the considered features are the features themselves, their squares and products of 2 taken at a time; similarly for the cubic combinations

- The code has been divided into generic functions. Context switching does make the code a bit inefficient, but it is easy to understand and debug in this way.
- The Mean Cube Error has been taken as :

$$\sum_{i=1}^{m} \left(h_\Theta(x^i) - y^i \right)^3 / 3m$$

- The plots are generated by the python scripts themselves and saved by names hard-coded in the script. The output will be shown on terminal, unless explicitly redirected to a file.

(a) Implementation of Linear Regression

(output/result in uploaded file: *output_a.txt*)

- Learned parameters in some cases (exhaustive data in the uploaded file):

Without regularization:
#iterations: 371
learned parameters: [[4.55219838e-09 4.29431183e-02 2.83732799e-02 4.50105722e-02 4.81126731e-01]]
MSE: 0.860117904218

With regularization:

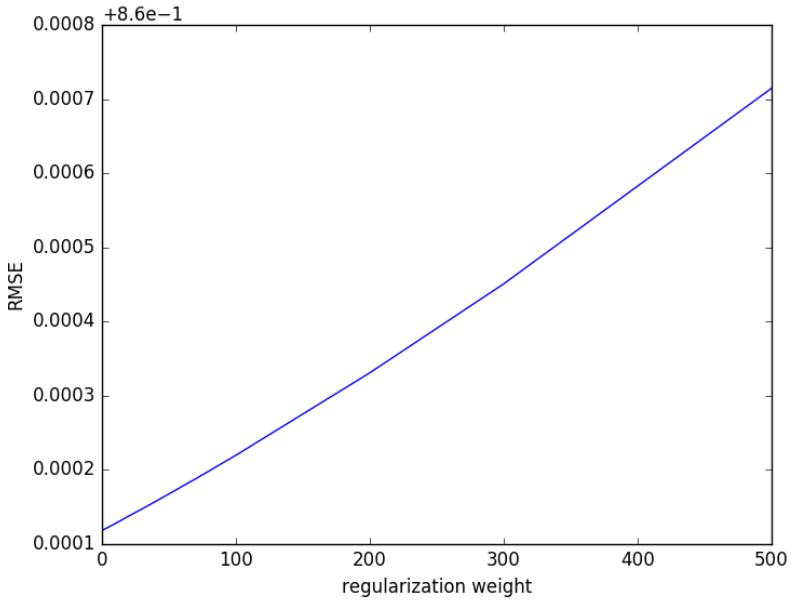
regularization param: 10
#iterations: 370
learned params: [[4.79178777e-09 4.29538732e-02 2.85549609e-02 4.51878018e-02 4.80662559e-01]]
MSE: 0.860127809653

regularization param: 50
#iterations: 368
learned params: [[5.30946012e-09 4.29939676e-02 2.92630823e-02 4.58774306e-02 4.78838131e-01]]
MSE: 0.860167583914

regularization param: 100
#iterations: 365
learned params: [[6.19269292e-09 4.30425461e-02 3.01362295e-02 4.67273290e-02 4.76579821e-01]]
MSE: 0.860219809126

regularization param: 500
#iterations: 345
learned params: [[1.72745775e-08 4.33523994e-02 3.65425783e-02 5.29371252e-02 4.59567850e-01]]
MSE: 0.860714573798

etc...



- Observations / Conclusions:

  The increase of RMSE with regulariztion in the observed graphs are, indeed surprising. However, it could be seen that the variations are within a very narrow band**.** So, it could be concluded that with this dataset, the regularization has **hardly any real effect** on the model performance.

  Infact, **the model performed slightly better without any regularization at all.**

(b) Experimenting with optimization algorithms

(output/result in uploaded file: *output_b.txt*)

- Some of the Learned parameters: (exhaustive data in uploaded file)

#iterations: 80

Gradient descent:
learned params: [[0.01563742 0.04964651 0.07437015 0.07760394 0.41647755]]
MSE: 0.868129061245

IRLS:
learned params: [[1.17093835e-16 4.28897509e-02 2.80575827e-02 4.46796089e-02 4.81659611e-01]]
MSE: 0.860094397137

#iterations: 200

Gradient descent:
learned params: [[2.18650386e-05 4.35441271e-02 3.18002884e-02 4.86516278e-02 4.75298894e-01]]
MSE: 0.8603864806

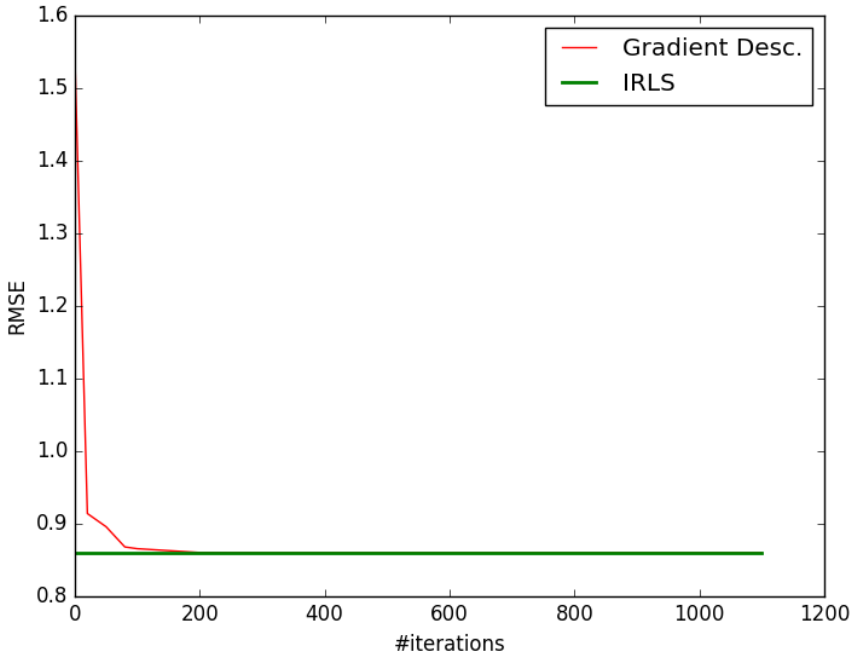
IRLS:
learned params: [[1.17093835e-16 4.28897509e-02 2.80575827e-02 4.46796089e-02 4.81659611e-01]]
MSE: 0.860094397137

#iterations: 500

Gradient descent:
learned params: [[4.79578519e-12 4.28951473e-02 2.80894962e-02 4.47130907e-02 4.81605722e-01]]
MSE: 0.860096761737

IRLS:
learned params: [[1.17093835e-16 4.28897509e-02 2.80575827e-02 4.46796089e-02 4.81659611e-01]]
MSE: 0.860094397137

etc...



- Observations / Conclusions:

  The RMSE for the Gradient Descent falls with the number of iterations and settles to a minimum at around 100 (for this data set). In general, this pattern would be followed – more the number of iterations, better the learning.

  The Iterative Re-weighted Least Square method, on the other hand, *for the linear regression problem* provides the least-error solution in single step (on MSE, which would, infact closely correspond to least RMSE as well). However, this is not, in general the case.

  So, for the case of Linear Regression, the IRLS algorithm is more computationally efficient.

(c) Experimenting with combinations of features

(output/result in uploaded file: *output_c.txt*)

- Some of the learned parameters: (exhaustive data in uploaded file)

  With Linear combination of features :

  Learning Rate: 0.001
  #iterations: 1639
  Learned Parameters: [[0.09753791 0.05981778 0.26768088 0.24988651 0.13911651]]

RMSE: 0.948360015486

Learning Rate: 0.005
#iterations: 127
Learned Parameters: [[0.04813984 0.07797956 0.06322357 0.0497283 0.40043638]]
RMSE: 0.87165921403

Learning Rate: 0.01
#iterations: 308
Learned Parameters: [[0.02865247 0.08382908 0.08618211 0.06307897 0.41459903]]
RMSE: 0.872869126636

Learning Rate: 0.1
#iterations: 67
Learned Parameters: [[2.24018082e-04 4.66863084e-02 4.94355218e-02 6.79359229e-02
4.44854115e-01]]
RMSE: 0.862176255269

etc...

With Quadratic Combinations:

Learning Rate: 0.001
#iterations: 2427
Learned Parameters: [[0.07228385 0.18785346 0.07281733 0.07528848 0.34446304 -0.15342381
0.05506447 -0.01295464 -0.02568324 0.108686 0.22046561 -0.14895971
0.08597997 0.05000792 0.15121356]]
RMSE: 0.862335059086

Learning Rate: 0.005
#iterations: 681
Learned Parameters: [[0.0063429 0.13077693 -0.00478217 0.07066573 0.39209965 -0.08176147
0.1319054 0.07310471 -0.19793407 0.1139071 0.20389994 -0.16988629
0.03447944 -0.21306675 0.46078087]]
RMSE: 0.842538268865

Learning Rate: 0.01
#iterations: 487
Learned Parameters: [[0.0010404 0.00423626 0.01916001 0.07372677 0.40544234 0.01602052
-0.03277621 -0.0416108 0.06364994 0.02834995 0.10448214 -0.04487797
-0.0011261 -0.05139185 0.25901044]]
RMSE: 0.833739802001

Learning Rate: 0.1
#iterations: 99
Learned Parameters: [[1.13519948e-05 3.08528671e-02 8.44432318e-02 6.50688649e-02
3.68034491e-01 -1.11597232e-03 2.70236180e-02 -2.11246734e-02
-1.99266841e-02 -1.51650335e-02 7.49367281e-02 -6.26577434e-02
-8.63925982e-03 -7.74926915e-02 3.39060998e-01]]
RMSE: 0.829679048783

With Cubic Combinations:

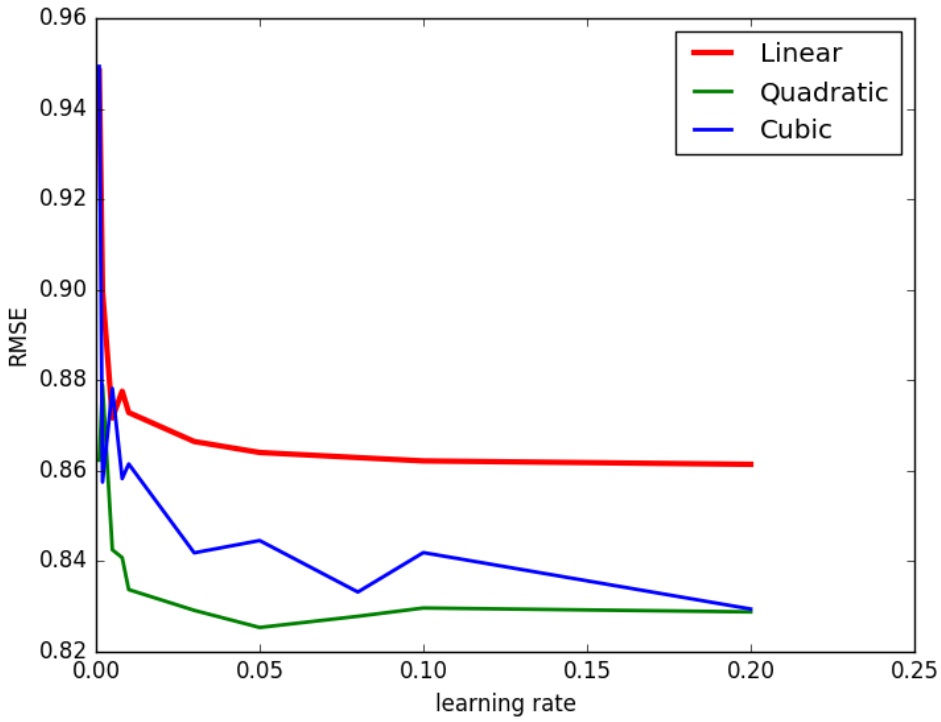
Learning Rate: 0.001
#iterations: 2689
Learned Parameters: [[0.03515875 0.16379676 0.12761472 -0.11278865 0.14711321 -0.46722977
-0.02680242 -0.10978569 0.12377088 -0.25864737 0.11049028 -0.33800661
0.3789243 -0.09131987 0.31998124 0.40186987 0.06305314 0.07232527
-0.01363717 0.09095721 -0.13680347 -0.17220446 0.15190985 0.21700241
-0.20118747 0.32203879 0.21814004 0.13904979 0.5656821 -0.31878314
0.19665646 0.35541594 0.03413469 -0.09683926 0.21321752]]

RMSE: 0.949345652995

Learning Rate: 0.005
#iterations: 1107
Learned Parameters: [[0.00207502 -0.1481562 0.06334365 0.13131068 0.36332343 0.22942724
0.12420367 -0.11510552 -0.13462771 -0.000834 0.02765945 -0.03238679
-0.05615672 0.00468004 0.32657283 -0.27294009 -0.13178176 0.18938691
0.08973073 0.21872288 -0.04342722 -0.23124063 0.20205918 -0.07124405
0.12326866 0.12564362 0.09579803 -0.14980062 0.07802443 0.04098931
-0.01934854 0.17693601 -0.11003323 -0.32336064 0.34594584]]
RMSE: 0.878176977692

Learning Rate: 0.1
#iterations: 226
Learned Parameters: [[3.02037415e-11 6.52287548e-02 1.53818654e-02 6.17825137e-02
3.88900165e-01 -6.08091743e-02 2.23543156e-02 -2.42316396e-02
3.70652959e-02 2.03493155e-02 6.28553590e-02 -1.23630057e-01
-2.93333684e-01 1.59263232e-02 3.36140260e-01 -3.96487729e-04
5.26651003e-02 5.18030763e-02 -6.20990447e-02 -2.55704842e-03
1.64658765e-02 -3.33129669e-02 9.98818307e-02 -1.46436883e-01
-5.83706819e-03 6.16896290e-02 1.31623135e-01 -1.15336651e-01
2.24641752e-01 -1.62650543e-01 2.51818866e-01 5.04150848e-01
-1.13129563e-01 -9.25200654e-02 5.92471956e-02]]
RMSE: 0.841918193371

etc...



- Observations / Conclusions:

In this case, the **Quadratic model** (taking quadratic combinations) of features gives least RMSE at low learning rates. Since it is advisable to keep learning rate around this range, using Quadratic combinations would be more efficient.
NOTE: In this case, the learning rates could have been extended further than 0.2. However, this has not been done because: (1) The learning rate is seldom taken more than 0.1 – anything more than this would be of little interest in general case. (2) The granularity and specifications of the graph at the lower (and more important) range would not have been visible.

(c) Experimenting with cost functions

(output/result in uploaded file: *output_d.txt*)

- Some of the learned parameters: (exhaustive data in uploaded file)

Using the Mean Squared Error cost function:

learning rate: 0.001
#iterations: 13146
learned parameters: [[1.01358442e-06 4.32796381e-02 3.03633607e-02 4.70937186e-02 4.77770154e-01]]
RMSE: 0.860271512734

learning rate: 0.005
#iterations: 3136
learned parameters: [[6.29844258e-08 4.30636358e-02 2.90843734e-02 4.57593154e-02 4.79923726e-01]]
RMSE: 0.860171482054

learning rate: 0.05
#iterations: 375
learned parameters: [[3.09535809e-09 4.29433377e-02 2.83745221e-02 4.50119731e-02 4.81124552e-01]]
RMSE: 0.860117991038

learning rate: 0.1
#iterations: 135
learned parameters: [[5.52862578e-07 4.29263810e-02 2.82709688e-02 4.49089347e-02 4.81294740e-01]]
RMSE: 0.860109975348

Using the Mean Absolute Error cost function:

learning rate: 0.001
#iterations: 7332
learned parameters: [[-0.15673271 0.04927556 0.05343523 0.05141795 0.30398966]]
RMSE: 0.883072885439

learning rate: 0.005
#iterations: 2716
learned parameters: [[-0.15651405 0.04785687 0.05299963 0.0506027 0.30514717]]
RMSE: 0.882907470749

learning rate: 0.05
#iterations: 172
learned parameters: [[-0.15646598 0.04803441 0.05275867 0.05011473 0.30589723]]
RMSE: 0.882792487262

learning rate: 0.1
#iterations: 142
learned parameters: [[-0.15646067 0.04793102 0.05285137 0.05016481 0.30578991]]
RMSE: 0.882808471736

Using the Mean Cube Error cost function:

learning rate: 0.001
#iterations: 93

learned parameters: [[0.67489921 0.27346952 0.75001136 0.21396838 -0.07854768]]
RMSE: 1.545936271

learning rate: 0.005
#iterations: 25
learned parameters: [[0.4643509 0.29993318 0.32592843 0.09717915 0.50684173]]
RMSE: 1.21565363171

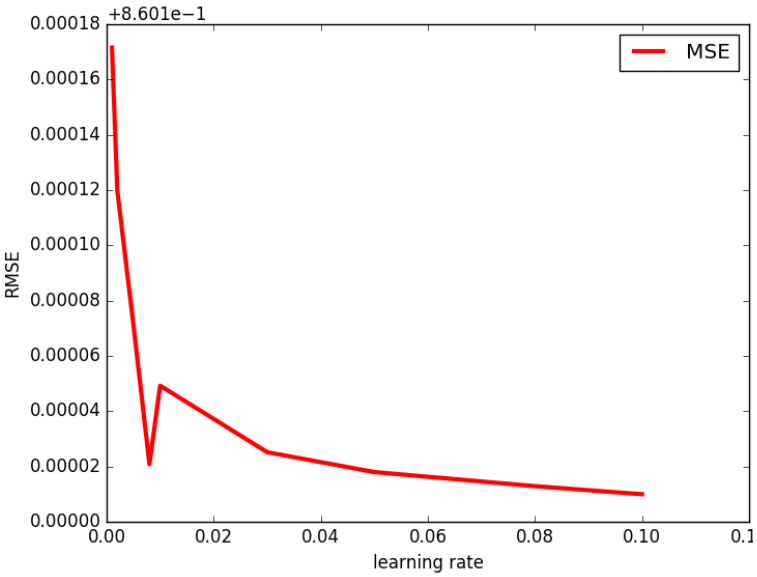
learning rate: 0.05
#iterations: 6
learned parameters: [[0.38003939 0.09517642 0.19165244 0.28299462 0.43450054]]
RMSE: 1.0614366671

learning rate: 0.1
#iterations: 2
learned parameters: [[0.03254287 0.11753648 0.54508504 0.47541407 0.18274039]]
RMSE: 1.1268996667

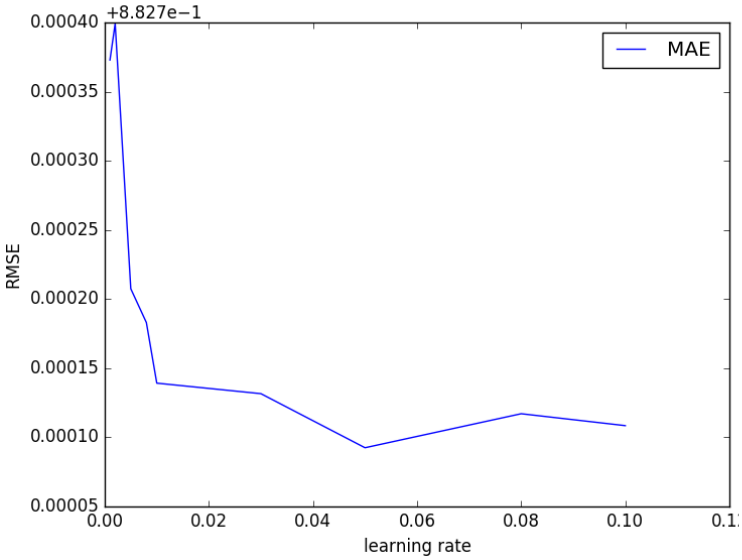
etc...

NOTE: Because of wide differences in the range over which the RMSE varies in the graphs (especially in the Mean Cubic error case), the graphs have been plotted differently:

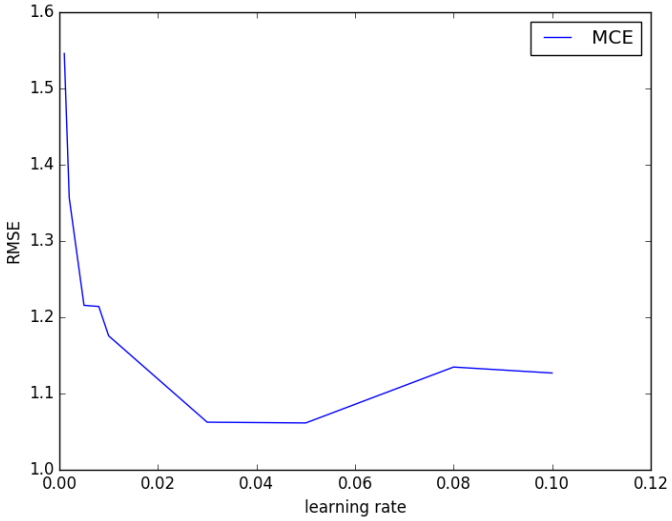
For Mean Square Error:



For Mean Absoulte Error:



For Mean Cube Error:



- Observations / Conclusions:

  Clearly, based on the observations from the graphs, the Mean Cube Error could be eliminated, as the RMSE of the MCE case is significantly more than the other 2 cost functions.

  Of the other 2, it could be easily observed that the graph for Mean Squared Error has lower axis multiplication factor (0.86) and also lower values on the axis. Resultantly, **Mean Squared Error could be said to be the best cost function to minimize in this case.**

  A reason behind this observation could be that the Mean Squared Error is a better indicator of the Root Mean Squared Error than the other 2 cost functions. Since the final result is being seen for the RMSE values, the MSE may have some advantage over other methods.