Amazon Salesrank Estimation Based on Network Lasso

Khashayar Kamran Kimia ShayestehFard

December 12, 2017

Incentive: Predicting Amazon SalesRank

.

Sales rank of an item on Amazon is a 1 to 8 digit number that indicates the popularity of an item. It captures the previous sale success of an item and how it is gonna sell in the future. Amazon calculates this number but does not disclose the algorithm. Knowing an item's sales rank would help Amazon sellers of all products answer this question: Should I buy, or should I pass? Or more accurately: Is there a demand for this product?

We have obtained a huge data set consist of Amazon Products from SNAP [1] project with about 160,000 different products (Books, music CDs, DVDs and VHS video tapes and etc). Each product is labeled with its unique ASIN as you can see on the Amazon website. The data set forms a graph where each node is a product and is connected to products which got co-purchased with that product. This co-purchased network is sparse and each product is usually connected to 0 to 5 products. There are categorical data and customers reviews available in the data set along with the sales rank of the products.

Problem Formulation : Network Lasso

.

The method used in this project is called **Network Lasso**. It focuses on optimization and clustering on the large graphs. The formulation usually looks like this:
Consider the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the vertex set and $\mathcal{E}$ the set of edges. Each node i represents a data point and the edges represent the connection of particular data point to each other.

$$\underset{\beta}{\text{minimize}} \sum_{i \in \mathcal{V}} f_i(\beta_i) + \sum_{(j,k) \in \mathcal{E}} g_{jk}(\beta_j, \beta_k) \tag{1}$$

Where f_i is the cost function at node i, and g_{jk} is the cost function associated with edge (j, k). An example of simple linear regression model with squared loss function and the convex problem definition would look like this:

$$\underset{\beta}{\text{minimize}} \sum_{i \in \mathcal{V}} ||\text{Salesrank}_i - \beta_i^T x_i||_2^2 + \mu ||\beta_i||_2^2 + \lambda \sum_{(j,k) \in \mathcal{E}} w_{jk} ||\beta_j - \beta_k||_2^2$$

Note that here, unlike the regular linear regression model, each data point has its own β_i and the model tries to jointly choose β_is for the whole training set in order to minimize the objective. So one question is that how to infer the value of β_j of a new node j for example in the test set? As mentioned in [2], after solving for β^*, we can interpolate the solution to estimate the value of β_j on a new node j, for example during cross-validation on a test set. Given j, all we need is its feature and its connections within the network. With this information, we treat j like a dummy node, with $f_j(\beta_j) = 0$. We solve for β_j just like in problem (1) except without the objective function f_j:

$$\text{minimize} \sum_{k \in N(j)} w_{jk} ||\beta_j - \beta_k^*||_2^2 \tag{2}$$

where $N(j)$ is the set of neighbors of node j.

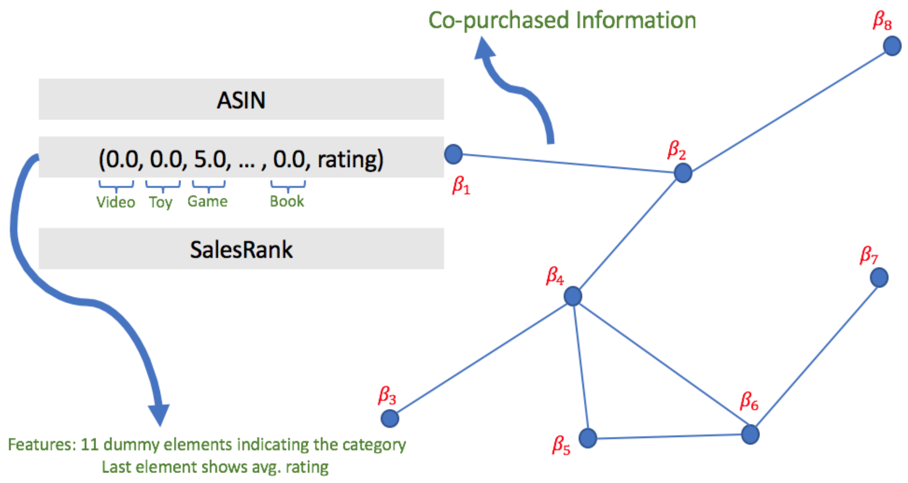
λ can be viewed as a single parameter which is tuned to yield different global results. λ defines a trade-off for the nodes between minimizing its own objective and agreeing with its neighbors. Let's look at two extreme values of λ. At $\lambda = 0$, β_i^* the solution at node i, is simply a minimizer of f_i. This can be computed locally at each node, since when $\lambda = 0$ the edges of the network have no effect. At the other extreme, as $\lambda \to \infty$, problem 1 turns into

$$\text{minimize} \sum_{i \in \mathcal{V}} f_i(\tilde{\beta})$$

where one should find a global identical $\tilde{\beta}$ for all the nodes. One can find the optimal value of λ using techniques like cross-validation.

Our approach

We are going to use a linear regression model in the network lasso setting to predict the sales rank of a product. We are going to use Apache Spark to solve this problem in a parallelize and distributed fashion using gradient descent with diminishing step size.



Feature Extraction:

We extract the following RDDs from our data set:

Salesrank RDD: An RDD consisting of pairs of (ASIN, Salesrank) where we divide the original salerank value to 100000 to be in range of [1-100] and be in the same order of other features.

Features RDD: An RDD consisting of pairs of (ASIN, x) where feature vector x is a 1×12 vector. The first 11 elements are dummy elements indicating the category group of the product. Based on the category of the product, one of these 11 elements is set to 5.0 and others are set to 0.0. The 12th feature is the average rating of the product.

Connection RDD: The connection information of the graph.

Objective Function and Steps of Algorithm:

We also assume all the weights on the links to be equal to 1. Our objective function looks like:

$$\text{minimize} F = \sum_{i \in \mathcal{V}} ||\text{Salesrank}_i - \beta_i^T x_i||_2^2 + \mu ||\beta_i||_2^2 + \lambda \sum_{(j,k) \in \mathcal{E}} ||\beta_j - \beta_k||_2^2$$

Using the gradient descent we update each β_i individually in each iteration as follows:

$$\beta_i^{(k+1)} = \beta_i^{(k)} - \alpha^{(k)} \nabla_{\beta_i}^{(k)} F$$

The question is when do we stop the iteration process? Since in this setting we have a large RDD of βs at each data point, we can not look at all the individual norm of gradients. So we use $max\{||\nabla_{\beta_i}^{(k)} F||_2^2\}$ for our convergence metric. We continue the iteration until $max\{||\nabla_{\beta_i}^{(k)} F||_2^2\} < \epsilon$ or we reach to a maximum number of iterations.

Gradients are calculated as:

$$\nabla_{\beta_i}^{(k)} F = -2(\text{Salesrank}_i - \beta_i^T x_i)x_i + 2\mu\beta_i + \lambda \sum_{k \in N(i)} 2(\beta_i - \beta_k) * 2$$

The last term is multiplied by 2 since we calculate it 2 times for each i: once when looking at its own neighbors and once when it appears on its neighbors links. We use a diminishing step size: $\alpha^{(k)} = \frac{\text{gain}}{k^{\text{pow}}}$

Testing

After finding β_i^* on the training set, for each test data point j that has at least one neighbor in the training set, we infer its beta according to problem 2:

$$\hat{\beta}_j = \frac{\sum_{k \in N(j)} \beta_k}{|N(j)|}$$

We use this β to predict the sales rank of a product. After that we can calculate the Total Test Error (TTE):

$$\text{Total Test Error} = \sum_{i \in \mathcal{V}_{\text{test}}} ||\text{Salesrank}_i - \hat{\beta}_i^T x_i||_2^2$$

Distributed Implementation:

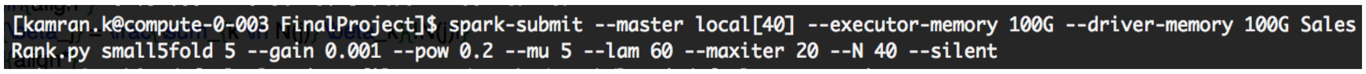
Looking at the problem formulation and the steps of the training algorithm, we can see that the problem can be solved in a distributed manner. Each β can be trained individually and independently of the whole network and the only information it needs is the value of β on its neighbors.

Parallelization:

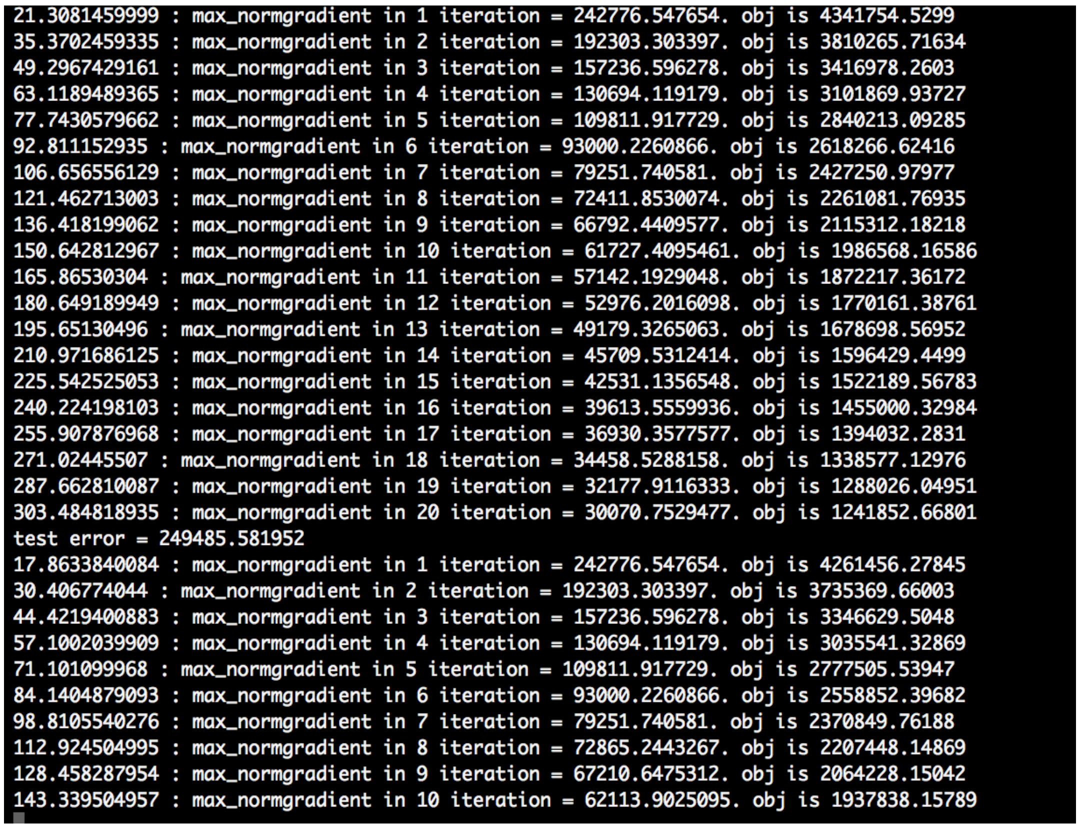
We can see that each step of the algorithm is performed on one data point and informations from its neighbors. Each node has limited neighbors and the network connection is sparse. Dimension of each feature is 12. So as the size of problem grows the step performed on each data point does not grow. So we can break the data into a large number of machines and partitions and be done in parallel. All the steps can be done using map, reduce and join.

Validation

We use 5-fold cross validation on our data set of size 160000 to test our approach. We pick a constant regularization factor $\mu = 5$ and change the value of λ to find the minimum total test error(TTE). You can see the other variables (gain, pow, N, memory, maxiteration and etc) and the convergence of the algorithm below:

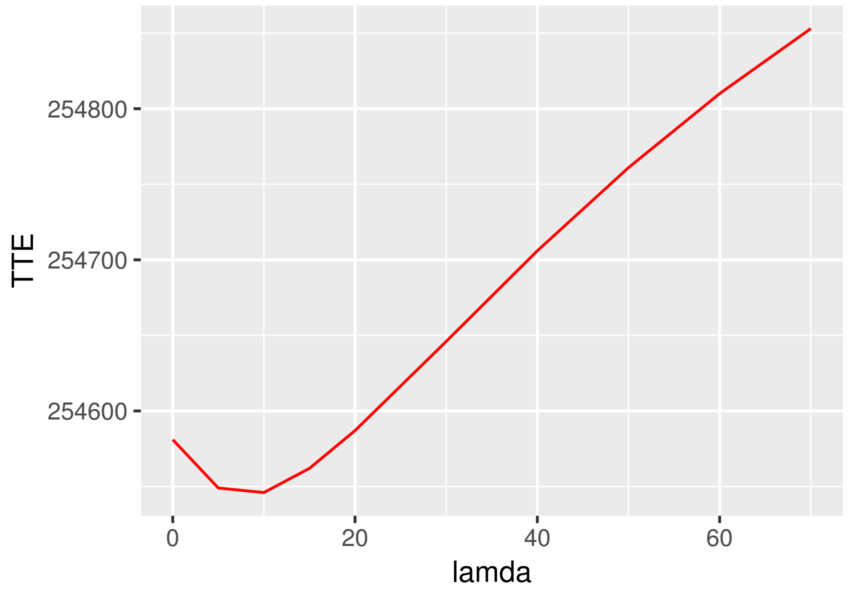


```
[kamran.k@compute-0-003 FinalProject]$ spark-submit --master local[40] --executor-memory 100G --driver-memory 100G Sales
Rank.py small5fold 5 --gain 0.001 --pow 0.2 --mu 5 --lam 60 --maxiter 20 --N 40 --silent
```

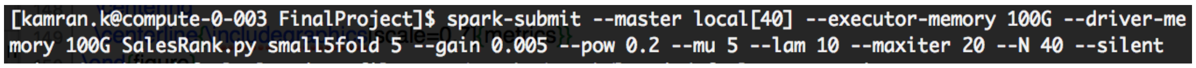


```
21.3081459999 : max_normgradient in 1 iteration = 242776.547654. obj is 4341754.5299
35.3702459335 : max_normgradient in 2 iteration = 192303.303397. obj is 3810265.71634
49.2967429161 : max_normgradient in 3 iteration = 157236.596278. obj is 3416978.2603
63.1189489365 : max_normgradient in 4 iteration = 130694.119179. obj is 3101869.93727
77.7430579662 : max_normgradient in 5 iteration = 109811.917729. obj is 2840213.09285
92.811152935 : max_normgradient in 6 iteration = 93000.2260866. obj is 2618266.62416
106.656556129 : max_normgradient in 7 iteration = 79251.740581. obj is 2427250.97977
121.462713003 : max_normgradient in 8 iteration = 72411.8530074. obj is 2261081.76935
136.418199062 : max_normgradient in 9 iteration = 66792.4409577. obj is 2115312.18218
150.642812967 : max_normgradient in 10 iteration = 61727.4095461. obj is 1986568.16586
165.86530304 : max_normgradient in 11 iteration = 57142.1929048. obj is 1872217.36172
180.649189949 : max_normgradient in 12 iteration = 52976.2016098. obj is 1770161.38761
195.65130496 : max_normgradient in 13 iteration = 49179.3265063. obj is 1678698.56952
210.971686125 : max_normgradient in 14 iteration = 45709.5312414. obj is 1596429.4499
225.542525053 : max_normgradient in 15 iteration = 42531.1356548. obj is 1522189.56783
240.224198103 : max_normgradient in 16 iteration = 39613.5559936. obj is 1455000.32984
255.907876968 : max_normgradient in 17 iteration = 36930.3577577. obj is 1394032.2831
271.02445507 : max_normgradient in 18 iteration = 34458.5288158. obj is 1338577.12976
287.662810087 : max_normgradient in 19 iteration = 32177.9116333. obj is 1288026.04951
303.484818935 : max_normgradient in 20 iteration = 30070.7529477. obj is 1241852.66801
test error = 249485.581952
17.8633840084 : max_normgradient in 1 iteration = 242776.547654. obj is 4261456.27845
30.406774044 : max_normgradient in 2 iteration = 192303.303397. obj is 3735369.66003
44.4219400883 : max_normgradient in 3 iteration = 157236.596278. obj is 3346629.5048
57.1002039909 : max_normgradient in 4 iteration = 130694.119179. obj is 3035541.32869
71.101099968 : max_normgradient in 5 iteration = 109811.917729. obj is 2777505.53947
84.1404879093 : max_normgradient in 6 iteration = 93000.2260866. obj is 2558852.39682
98.8105540276 : max_normgradient in 7 iteration = 79251.740581. obj is 2370849.76188
112.924504995 : max_normgradient in 8 iteration = 72865.2443267. obj is 2207448.14869
128.458287954 : max_normgradient in 9 iteration = 67210.6475312. obj is 2064228.15042
143.339504957 : max_normgradient in 10 iteration = 62113.9025095. obj is 1937838.15789
```

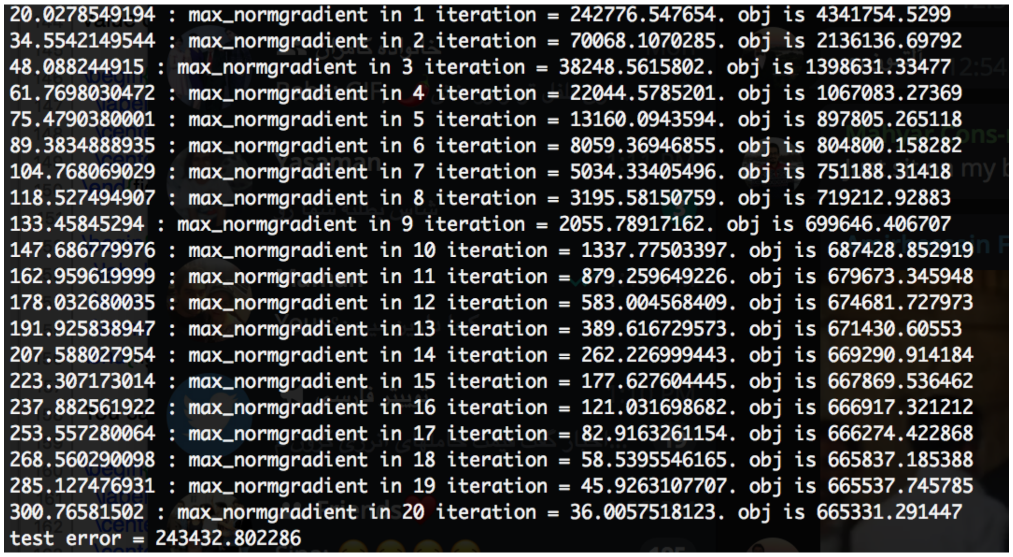
You can see the Total Test Error results for $\mu = 5$ below:



The minimum Total Test Result is achieved at $\lambda = 10$. In small values of λ and μ such as the achieved optimal point of $\lambda = 10, \mu = 5$ you can use larger gain to achieve faster convergence.



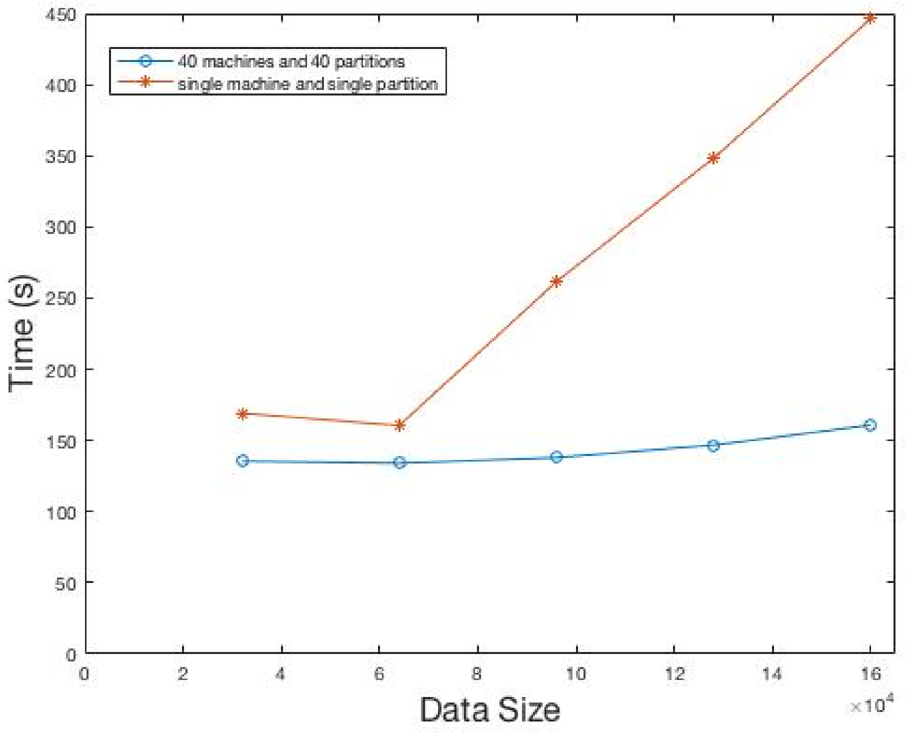
```
[kamran.k@compute-0-003 FinalProject]$ spark-submit --master local[40] --executor-memory 100G --driver-me
mory 100G SalesRank.py small5fold 5 --gain 0.005 --pow 0.2 --mu 5 --lam 10 --maxiter 20 --N 40 --silent
```



```
20.0278549194 : max_normgradient in 1 iteration = 242776.547654. obj is 4341754.5299
34.5542149544 : max_normgradient in 2 iteration = 70068.1070285. obj is 2136136.69792
48.088244915 : max_normgradient in 3 iteration = 38248.5615802. obj is 1398631.33477
61.7698030472 : max_normgradient in 4 iteration = 22044.5785201. obj is 1067083.27369
75.4790380001 : max_normgradient in 5 iteration = 13160.0943594. obj is 897805.265118
89.3834888935 : max_normgradient in 6 iteration = 8059.36946855. obj is 804800.158282
104.768069029 : max_normgradient in 7 iteration = 5034.33405496. obj is 751188.31418
118.527494907 : max_normgradient in 8 iteration = 3195.58150759. obj is 719212.92883
133.45845294 : max_normgradient in 9 iteration = 2055.78917162. obj is 699646.406707
147.686779976 : max_normgradient in 10 iteration = 1337.77503397. obj is 687428.852919
162.959619999 : max_normgradient in 11 iteration = 879.259649226. obj is 679673.345948
178.032680035 : max_normgradient in 12 iteration = 583.004568409. obj is 674681.727973
191.925838947 : max_normgradient in 13 iteration = 389.616729573. obj is 671430.60553
207.588027954 : max_normgradient in 14 iteration = 262.226999443. obj is 669290.914184
223.307173014 : max_normgradient in 15 iteration = 177.627604445. obj is 667869.536462
237.882561922 : max_normgradient in 16 iteration = 121.031698682. obj is 666917.321212
253.557280064 : max_normgradient in 17 iteration = 82.9163261154. obj is 666274.422868
268.560290098 : max_normgradient in 18 iteration = 58.5395546165. obj is 665837.185388
285.127476931 : max_normgradient in 19 iteration = 45.9263107707. obj is 665537.745785
300.76581502 : max_normgradient in 20 iteration = 36.0057518123. obj is 665331.291447
test error = 243432.802286
```

Parallelism Performance

As explained in parallelization analysis part we expect that on large data sets, breaking the data set to a large number of partitions on different machines be time-wise beneficial. We run our algorithm on a single core with one partition and compare its run time to when we run it on 40 core with 40 partitions. We repeat this by increasing the size of the data set to see the effect of parallelism and breaking the data into partitions. We used compute node **Compute-0-087** and below you can see the result:



40 machines and 40 partitions
single machine and single partition
Time (s)
Data Size
$\times 10^4$
0
50
100
150
200
250
300
350
400
450
0
2
4
6
8
10
12
14
16

Bibliography

[1] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection, June 2014.

[2] David Hallac, Jure Leskovec, and Stephen Boyd. Network lasso: Clustering and optimization in large graphs. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 387–396, New York, NY, USA, 2015. ACM.