Sri Gautham Subramani

Comparison of Recommendation Systems on Amazon Review DataSet

Summary

The purpose of this project is to understand the advantages and disadvantages of various recommendation systems, thus in turn understanding how the recommendation systems are chosen for various use cases in the industry. This project is heavily influenced by the Netflix Prize open competition. We chose Amazon open dataset, as it is already collected and is readily available for research. We got the data from http://jmcauley.ucsd.edu/data/amazon/. This dataset contains product reviews and metadata from Amazon, including 142.8 million reviews spanning May 1996 – July 2014. The product reviews data contains information about ratings, text, helpfulness votes and the metadata contains information about descriptions, category information, price, brand and image features.

Introduction

From Amazon recommending products you may be interested in based on your recent purchases to Netflix recommending shows and movies you may want to watch; recommender systems have become popular across many applications of data science. Like many other problems in data science, there are several ways to approach recommendations. Two of the most popular are collaborative filtering and content-based recommendations. In Collaborative Filtering, for each user, recommender systems recommend items based on how similar users liked the item. In Content-based recommendation, recommender systems work based on the detailed metadata about each item. We build an item profile for each item and based on this item profile, recommendations are provided.

We have explored various recommendation systems in both Content-based recommendation and Collaborative Filtering. The recommendation techniques tried and tested are: mean, median, baseline, Un-Weighted Cosine Similarity, Weighted Cosine Similarity, K-Nearest Neighbors (KNN), Single Value Decomposition (SVD), Alternative Least Squares (ALS), Dimension Independent Matrix Square using MapReduce (DIMSUM). Let us explore the Root Mean Square Error (RMSE) for each of the above-mentioned algorithms, as it aids in easier comparison of effective algorithms. Let's have a look on the structure of the input data.

	productID	reviewerID	overall
0	0000013714	ACNGUPJ3A3TM9	4.0
1	0000013714	A2SUAM1J3GNN3B	5.0
2	0000013714	APOZ15IEYQRRR	5.0
3	0000013714	AYEDW3BFK53XK	5.0
4	0000013714	A1KLCGLCXYP1U1	3.0

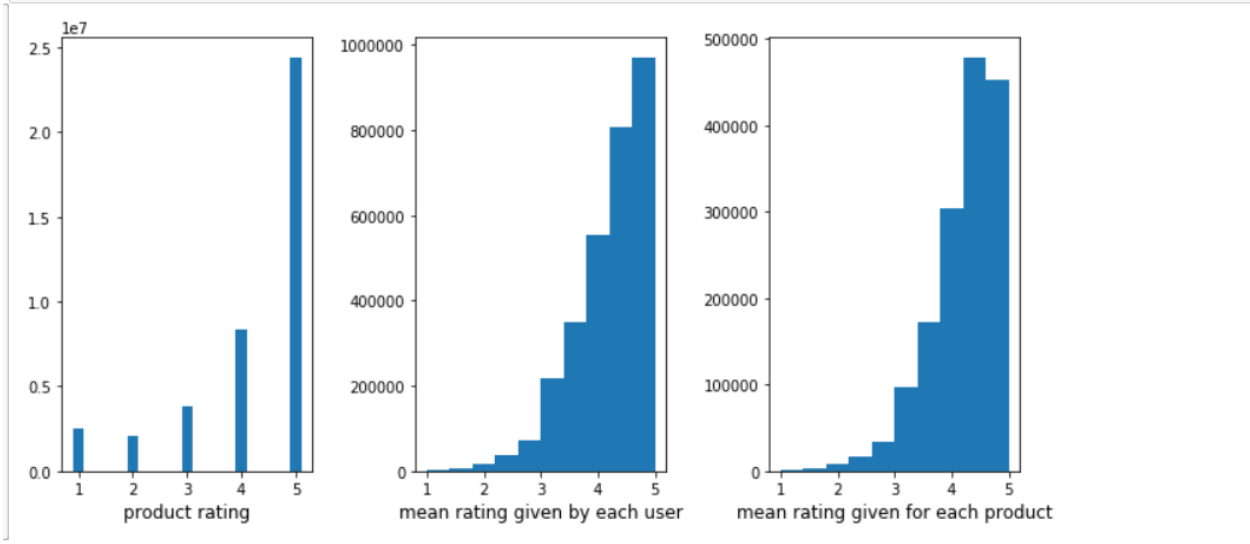
In our data most of the products have either no rating or only one rating and most of the users have given no rating or a maximum of one rating. With these data we get to a cold start problem. We don't have enough data to give predictions to these users and items. Therefore, on a safer side we took only the users whom have given a minimum of 5 ratings and the items which have a minimum of 5 ratings. We call these as 5-core data. Overall, we have 41 million such ratings.

```
print("Number of unique products =", len(df['productID'].unique()))
print("Number of unique users =", len(df['reviewerID'].unique()))
print("Number of ratings =", len(df))
```

```
Number of unique products = 1569973
Number of unique users = 3035045
Number of ratings = 41135700
```

Mean, Median

Before we start working on the data, it's important that we analyze the data and have an overview about them. Let us have some statistical overview of all the ratings. As we can see in the below figure, most of the ratings are 5/5. There has been a study on the psychological behavior of a human on rating an item. A person by default gives a rating of 5 even if he/she is not highly satisfied, they give lesser ratings only if they are unsatisfied with the item. That's why in some cases the ratings are fixed on a scale of -2 to 2; 0 being the baseline. It gives us better scale to work on our recommendations. Also, the number of users who have given a mean rating of 5 is higher compared to the other users and it holds same for items as well with higher number of items with mean rating of about 5.



A simpler and easier way to implement recommendation system is to give recommendations based on the mean and/or median rating values. We calculate the mean/median of all the ratings that we have in our system and give that value as the predicted rating for unknown user – item pairs. It's evident to assume that this method of recommendation would not fair well.

```
mean_score = df['overall'].mean()
median_score = df['overall'].median()

print("Mean Score ", mean_score)
print("Median Score ", median_score)
```

```
Mean Score  4.215769635620641
Median Score  5.0
```

The mean score of all the ratings is 4.21 and the median score is 5. That means more than 50% of the items are rated 5. When we use these scores as the predicted rating score for all the user-item pair, then we get the Root Mean Square Error (RMSE) value as follows:

```
mean_rmse_benchmark = np.sqrt(pow(df['overall'] - mean_score, 2).mean())
print("Root-mean-square error for mean score = {:4.2f}".format(mean_rmse_benchmark))

median_rmse_benchmark = np.sqrt(pow(df['overall'] - median_score, 2).mean())
print("Root-mean-square error for median score = {:4.2f}".format(median_rmse_benchmark))
```

```
Root-mean-square error for mean score = 1.18
Root-mean-square error for median score = 1.42
```

As we can see the RMSE value is very high which is not a good recommendation system. But it gives us a good baseline score to compare and see how the other recommendation systems perform.

Cosine Similarity – Unweighted

Cosine similarity is a measure of similarity between two non-zero vectors of an inner product space that measures the cosine of the angle between them. When two vectors are same we get a cosine similarity value of 1 as the angle between them is 0^o and if the vectors are completely different from each other, then the cosine similarity value is 0 as the angle between them is 90^o. The vectors are similar if they are parallel and dissimilar if they are perpendicular to each other. The cosine similarity between user and user is the normalized dot product of their row vectors, A and B respectively in the rating matrix R,

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

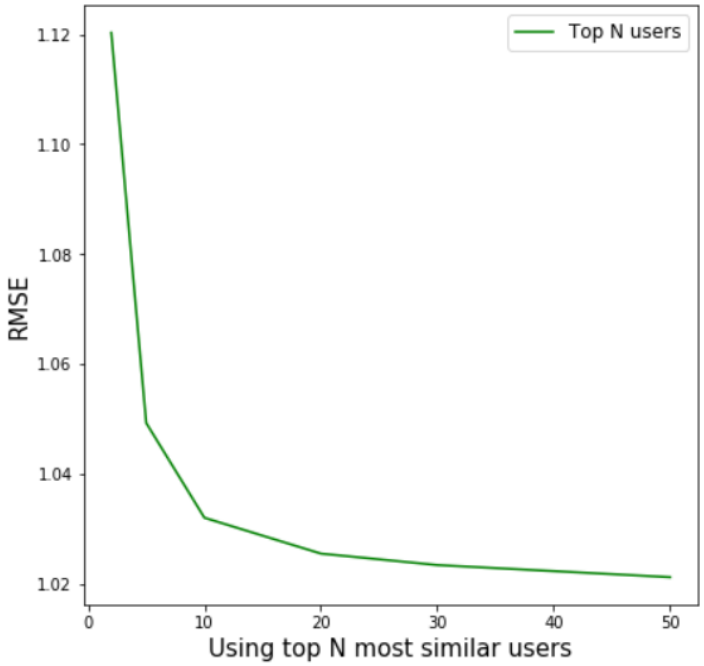
The Root mean square error we get using this unweighted cosine similarity method is, 1.23. This means that for all the ratings that we predict, we are still off by 1 rating. This is still bad, we would like to get as close as the original rating. If we compare this with the baseline score of mean, which is 1.18, we get a poor performance.

Cosine Similarity – Weighted

The major drawback of using unweighted cosine similarity is that, we give equal weight to all the items while calculating the similarity. We can make this work better by giving more weight to the most similar items, this way we can achieve higher efficiency. We get RMSE score of 1.02 for this method. We can see a considerable improvement in the RMSE score.

Top – N Similar Ratings

We could consider only N most similar items while predicting rating for an item. When we compare by taking different value for N in top N most similar items, we get different RMSE values. By plotting them into a graph, we can see that the RMSE value reduces exponentially till 20 most similar items and then it stretches out to a straight line. Taking the 20 most similar items gives us a better RMSE value with good performance trade-off, which has a RMSE value of 1.025.



Singular Value Decomposition (SVD)

In linear algebra, the Singular-valued decomposition is a factorization of a real or complex matrix. It gives us values in the order of most significant components, so that we can ignore the least significant components. By splitting the input data into 5 folds of training data, we get the RMSE value as follows. The mean RMSE value that we got with SVD method of recommendation is 1.06.

```
Evaluating RMSE, MAE of algorithm SVD on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.0952  1.0532  1.0668  1.0658  1.0619  1.0686  0.0142
MAE (testset)     0.8382  0.8145  0.8264  0.8220  0.8209  0.8244  0.0079
Fit time          1.97    1.98    1.99    1.97    2.20    2.02    0.09
Test time         0.05    0.05    0.05    0.05    0.05    0.05    0.00
```

By splitting the input data into train and test data set and into 3 Folds, we get the RMSE value as follows:

```
kf = KFold(n_splits=3)
algoSVD2 = SVD()

for trainset, testset in kf.split(data):
    # train and test algorithm.
    algoSVD2.fit(trainset)
    predictions = algo.test(testset)

    # Compute and print Root Mean Squared Error
    accuracy.rmse(predictions, verbose=True)
```

```
RMSE: 0.8226
RMSE: 0.8223
RMSE: 0.8301
```

The predicted rating for the user “0000013714” and item “A3W2PX96K1BA3M” is 4.486.

```
algoSVD.predict("0000013714", "A3W2PX96K1BA3M", 4)
```

```
Prediction(uid='0000013714', iid='A3W2PX96K1BA3M', r_ui=4, est=4.486313133208141,
```

K – Nearest Neighbors

KNNs are like top-N cosine similarities, they calculate the similarity and take the k-most similar items for recommendation. The SURPRISE package gives us an easier way of implementing KNN recommendation systems using cosine similarity. We divide the data into 2 parts: train and test. We fit the KNN model using the train data and then use the trained model on the test data to make predictions. Predicted ratings for few user – item pairs are as follows:

```
print(predictions[1:100])
```

```
[Prediction(uid='0000013714', iid='A3W2PX96K1BA3M', r_ui=4.232742857142857, est=4.232742857142857, details={'reason': 'Not eno
ugh neighbors.', 'was_impossible': True}), Prediction(uid='0000013714', iid='A2GKR2Q7MD8DG4', r_ui=4.232742857142857, est=4.23
2742857142857, details={'reason': 'Not enough neighbors.', 'was_impossible': True}), Prediction(uid='0000013714', iid='A1MC4E0
0RO5E9T', r_ui=4.232742857142857, est=4.232742857142857, details={'reason': 'Not enough neighbors.', 'was_impossible': True}),
Prediction(uid='0000013714', iid='A23PISU0ZLW71C', r_ui=4.232742857142857, est=4.232742857142857, details={'reason': 'Not enou
gh neighbors.', 'was_impossible': True}), Prediction(uid='0000013714', iid='A2G0LNLN79Q6HR', r_ui=4.232742857142857, est=4.232
742857142857, details={'reason': 'Not enough neighbors.', 'was_impossible': True}), Prediction(uid='0000013714', iid='A2R3K1KX
09QBYP', r_ui=4.232742857142857, est=4.232742857142857, details={'reason': 'Not enough neighbors.', 'was_impossible': True}),
Prediction(uid='0000013714', iid='A1P0IHU93EF9ZK', r_ui=4.232742857142857, est=4.232742857142857, details={'reason': 'Not enou
gh neighbors.', 'was_impossible': True}), Prediction(uid='0000013714', iid='A1KLRMWW2FWPL4', r_ui=4.232742857142857, est=4.232
742857142857, details={'reason': 'Not enough neighbors.', 'was_impossible': True}), Prediction(uid='0000013714', iid='A1GQPAM8
Y45QN7', r_ui=4.232742857142857, est=4.232742857142857, details={'reason': 'Not enough neighbors.', 'was_impossible': True}),
Prediction(uid='0000013714', iid='A2G5TCU2WDFZ65', r_ui=4.232742857142857, est=4.232742857142857, details={'reason': 'Not enou
```

Using this model, we can get top N recommended items for each user. The user “0000013714” would get “A14A5Q8VJK5NLR”, “A3W2PX96K1BA3M” and “A2GKR2Q7MD8DG4” as the 3 recommendations based on his rating history.

```
get_topN_recommendations(predictions, 3)
```

```
defaultdict(list,
            {'0000013714': [('A14A5Q8VJK5NLR', 4.232742857142857),
              ('A3W2PX96K1BA3M', 4.232742857142857),
              ('A2GKR2Q7MD8DG4', 4.232742857142857)],
             '0000029831': [('ACNGUPJ3A3TM9', 4.232742857142857),
              ('A2SUAM1J3GNN3B', 4.232742857142857),
              ('APOZ15IEYQRRR', 4.232742857142857)],
             '0000031887': [('A1K1JW1C5CUSUZ', 5),
              ('A1Z54EM24Y40LL', 5),
              ('A6HXFDIC7DVTC', 5)],
```

Slope-One

It is a simple yet accurate Collaborative Filtering recommendation technique, which drastically reduces overfitting, improve performance and ease implementation. Essentially, instead of using linear regression from one item's ratings to another item's ratings $[f(x) = ax + b)]$, it uses a simpler form of regression with a single free parameter $[f(x) = x + b]$. The free parameter is then simply the average difference between the two items' ratings. It was shown to be much more accurate than linear regression in some instances, and it takes half the storage or less. For better understanding, let us consider the below scenario:

Customer	Item A	Item B	Item C
John	5	3	2
Mark	3	4	Didn't rate it
Lucy	Didn't rate it	2	5

In this case, the average difference in ratings between item B and A is (2+(-1))/2=0.5. Hence, on average, item A is rated above item B by 0.5. Similarly, the average difference between item C and A is 3. Hence, if we attempt to predict the rating of Lucy for item A using her rating for item B, we get 2+0.5 = 2.5. Similarly, if we try to predict her rating for item A using her rating of item C, we get 5+3=8.

If a user rated several items, the predictions are simply combined using a weighted average where a good choice for the weight is the number of users having rated both items. In the above example, both John and Mark rated items A and B, hence weight of 2 and only John rated both items A and C, hence weight of 1 as shown below. we would predict the following rating for Lucy on item A as:

$$\frac{2 \times 2.5 + 1 \times 8}{2 + 1} = \frac{13}{3} = 4.33$$

Hence, given n items, to implement Slope One, all that is needed is to compute and store the average differences and the number of common ratings for each of the n^2 pairs of items.

```
kf = KFold(n_splits=3)
SlopeOnealgo = SlopeOne()

for trainset, testset in kf.split(data):
    # train and test algorithm.
    SlopeOnealgo.fit(trainset)
    predictions = SlopeOnealgo.test(testset)

    # Compute and print Root Mean Squared Error
    accuracy.rmse(predictions, verbose=True)
```

```
RMSE: 1.1453
RMSE: 1.1469
RMSE: 1.1289
```

Dimension Independent Matrix Square using MapReduce (DIMSUM)

The main disadvantage with Cosine Similarity recommendation systems is the complexity. It's of the order $O(n^2)$, which means, if we are to use the recommendation system on a dataset of a million, it would take up to a billion complexity. That's where DIMSUM comes into place, it uses MapReduce functionality in effectively calculating the Cosine Similarity. DIMSUM, an efficient and accurate all-pair similarity algorithm for real-world large-scale dataset, tackles shuffle size problem of several similarity measures using MapReduce. The algorithm uses a sampling technique to reduce `power items' and preserves similarities.

Algorithm 3 DIMSUMMapper(r_i)

for all pairs (a_{ij}, a_{ik}) in r_i **do**

With probability

$$\min\left(1, \gamma \frac{1}{||c_j||||c_k||}\right)$$

emit $((c_j, c_k) \rightarrow a_{ij}a_{ik})$

end for

Algorithm 4 DIMSUMReducer($(c_i, c_j), \langle v_1, \ldots, v_R \rangle$)

if $\frac{\gamma}{||c_i||||c_j||} > 1$ **then**

output $b_{ij} \rightarrow \frac{1}{||c_i||||c_j||} \sum_{i=1}^{R} v_i$

else

output $b_{ij} \rightarrow \frac{1}{\gamma} \sum_{i=1}^{R} v_i$

end if

We usually pick, $\gamma = 4\,log(n)/s$, where s is our desired similarity threshold. We see below the cosine similarity between different items.

Choosing every single column of A for the product leading to a column of A^TA, is often not required especially when the matrix A is very sparse. Making the decision to pass on a column of A, as a stochastic one allows for gains to be made in this computation. The main insight that allows gains in efficiency is sampling columns that have many non-zeros with lower probability. On the flip side, columns that have fewer non-zeros are sampled with higher probability. This

sampling scheme can be shown to provably accurately estimate cosine similarities, because those columns that have many non-zeros have more trials to be included in the sample, and thus can be sampled with lower probability.

```
dimsumSim.entries.take(10)
```

```
[MatrixEntry(656, 696, 0.22070377226756643),
 MatrixEntry(68, 111, 0.008702518557846753),
 MatrixEntry(133, 185, 0.03351579845910721),
 MatrixEntry(1, 354, 0.0010249521012127125),
 MatrixEntry(19, 37, 0.1184806933609093),
 MatrixEntry(81, 752, 0.05810905840484827),
 MatrixEntry(265, 274, 0.061014186837638575),
 MatrixEntry(74, 166, 0.014327826471018686),
 MatrixEntry(133, 391, 0.053350977640947934),
 MatrixEntry(35, 379, 0.022786907451052583)]
```

Alternating Least Square Errors (ALS)

Alternating Least Square Errors is a Collaborative Filtering based matrix factorization technique. We use the Machine Learning technique to factorize the rating matrix. We fix relatively a small value k and summarize each user with a k dimensional vector. Then to predict user u's rating for item i, we simply predict r_{ui} by approximating it to $x_u^T y_i$. We can formulate this as an optimization problem in which we aim to minimize the least squared error of the observed ratings. This optimization problem can be given as:

$$\min_{X,Y} \sum_{r_{ui}\, observed} (r_{ui} - x_u^{\intercal} y_i)^2 + \lambda(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2)$$

We got the best performance with rank 20 and iteration 15, however, with higher rank and higher iteration we get low performance.

By comparing all the above recommendation systems, by far we get lower RMSE value for ALS recommendation system. This gives us a good start to know about various recommendation systems and their performance on various parameters.

Future Work – Serendipity

Over the past several years, research in recommender systems has emphasized the importance of serendipity, but there is still no consensus on the definition of this concept and whether serendipitous items should be recommended is still not a well-addressed question. According to the most common definition, serendipity consists of three components: relevance, novelty and unexpectedness, where each component has multiple variations. There are two main reasons for

collaborative recommender systems to suggest serendipitous items: they broaden user preferences and increase user satisfaction.

Researchers often indicate that serendipitous items are very rare. However, it is unclear exactly how rare these items are, as it might not be worth of suggesting them due to their rareness and a high risk of suggesting irrelevant items, while optimizing for serendipity. An item is relevant to a user if the user expresses or will express their preference for the item in the future by liking or consuming the item depending on the application scenario. Novelty of an item to a user depends on how familiar the user is with the item. An item can be novel to a user in different ways:

- The user has never heard about the item.
- The user has heard about the item, but has not consumed it.
- The user has consumed the item and forgot about it.

Studies on serendipity in recommender systems often neglect the definition of unexpectedness. We present many definitions corresponding to the component. An item can be unexpected to the user if:

- The user does not expect this item to be relevant to them.
- The user does not expect this item to be recommended to them.
- The user would not have found this item on their own.
- The item is significantly dissimilar to items the user usually consumes.
- The user does not expect to find this item, as the user is looking for other kinds of items.

We plan to implement serendipity in our recommendation system by using the Linked Open Data. Generally, serendipity would work in a way that if a user likes Data Mining, the recommendation system generates the below link and in turn recommends HMM to the user.

Data Mining -> Machine Learning -> NLP -> Topic Modeling -> HMM

Serendipity is a very interesting and challenging task in recommendation systems and we would like to explore more about them in future.