# Udacity Machine Learning
# Final Project

John Ben Snyder

April 9, 2018

**Abstract**

Recommendation engines are a key component of online marketplaces, media services, and social media platforms. How does performance compare between the popular algorithms for such systems? In this paper, I use 9 million books ratings from Amazon users to test five different recommendation algorithms. I first set a baseline by splitting the data into training and test sets, and using the training from the mean to predict values on the test set. I then use the training set to build models using linear regression, collaborative filtering, singular value decomposition, restricted Boltzmann machines, and autoencoders to try to outperform the baseline. The project also involves representing the data in sparse forms that are tractable with a reasonable amount of computer memory. I find that singular value decomposition both performs best in terms of computer performance, and accuracy.

## 1 Project Overview

### 1.1 Background

Recommender systems play an essential role in online services. Netflix uses an ensemble of algorithms to estimate which films users will enjoy based on their past ratings. Spotify and Pandora determine what songs to play based on the past songs a user has liked or disliked. Online companies are well aware that leveraging their user's data can result in greater usage and sales. In fact, companies are willing to invest large resources for even modest improvements to their recommendation systems. For example, in 2009 Netflix awarded $1,000,000 for the development of a new ensemble algorithm for the movie recommender.

In the mid 1990s, online newsgroups began using an early form of collaborative filtering known as Tapestry. Tapestry was a relatively simple filtering system, where users could define tags and fellow users of interest. For example, a user could say they want articles tagged as "funny" by users "Jason, Katy, and Kim." Soon after, Amazon followed Tapestry's model in the early recommendation system. Because of the size of Amazon's user and item base (even early in the company's history), a simple filtering system wouldn't work. Instead, Amazon developed a method of measuring user and item similarity. This system looks for two users with similar purchase histories and ratings, and recommends new items based on what the other user has purchased. Since then, this collaborative filtering mechanism has been at the heart of recommendation systems. More recently, several new recommendation methods have appeared. Singular value decomposition, and deep learning techniques such as restricted Boltzmann machines and auto-encoders are used to generate some set of latent features describing the underlying data generation process.

### 1.2 Project Description

This project tests these different algorithms on one dataset, that is fairly representative of the general problem of recommendation systems. Using a dataset of 9 million Amazon user book reviews, I compare the performance, in terms of both computational efficiency and accuracy, of five recommendation methods.

The goal is the develop an algorithm that can predict a user's rating of some new item while minimizing the error to the user's actual rating of that item. As such, the dataset is split into

training and test sets, using 90% of the reviews to train a model, and 10% to measures the model's performance on new ratings. Consistent training and test sets are used across all 5 algorithms.

One of the more difficult problems with these data is just their size. While the dataset itself is not that large, these algorithms are built around the idea of having a matrix of user on one axis, and ratings on the other. This means dealing with large sparse matrices. In this case, the resulting matrix would be far too large (around 4 terabytes) to fit in memory on a mainstream computer. Therefore, a major portion of this project involves designing pipelines that can to efficiently handle the data in memory, while keeping processing time to something reasonable. I will cover these techniques in the methods section.

### 1.3 Metrics

Metrics for this project are relatively straightforward. For a given user, each algorithm returns a vector of ratings. The ratings for which the test dataset has values for that user are extracted from the vector. Once all ratings are extracted for all users, the predicted values from the algorithm are compared to the actuals in the test dataset using the root mean squared error.

$$RMSE = \sqrt{\frac{1}{N}\Sigma(\hat{y} - y)^2}$$

This metric is relatively simple, commonly used, and gives us an average deviation from the actuals, which is exactly what we're interested in.

I also report the computational time of each algorithm. Over several rounds of analysis the computation time differs by several times, and therefore can only reported reliably to an order of magnitude (seconds vs minutes vs hours). This might sound too broad, but the difference in performance between algorithms is on this scale, and is therefore the most useful measure. Computations were performed with an AMD FX 8320 with 16 GB of ram and a GTX 1070 GPU (used only for neural networks), running Ubuntu 16.04.

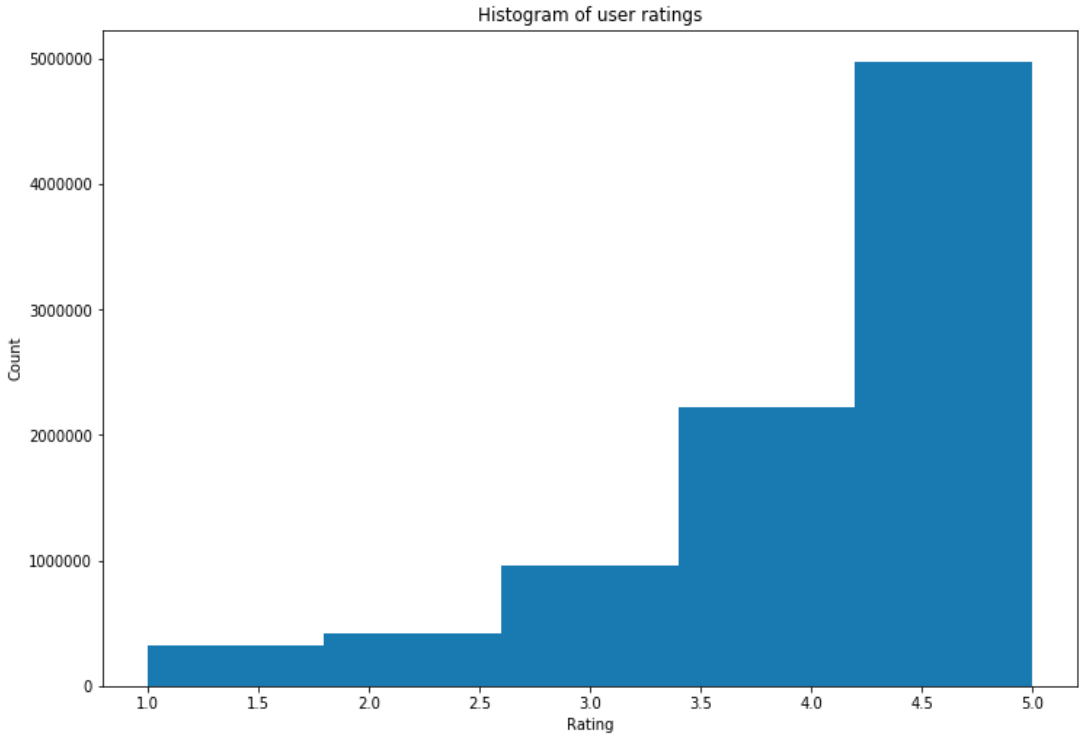
## 2 Analysis

The book review data were collected by Julian McAuley of the University of California San Diego. The data can be found here, as well as a number of other Amazon datasets. The available datasets cover most of Amazon's major product categories. For this project, however, I restrict the data to only the book reviews. I also restrict the data to the 5-core set. This means only users who have reviewed at least 5 books are included. This helps avoid the problem of an already sparse dataset being too sparse to make meaningful recommendations. For example, a user who has only reviewed a single book wont be much use for this project, as that single review must be in either the training or test data. If it is in the training data, it gives us no predictive ability, since the users similarity to other users wont correlate to other items. If the review is in the test data, there is nothing in the training data that can meaningfully predict the user, other than guessing the item's mean value.
The data are initially in the form of a single large json file. Each review includes an item id, helpfullness of the review, overall rating, text of review, time, summary, reviewer name, and id. two example entries are given below.


```
{'asin': '000100039X',
  'helpful': [0, 0],
  'overall': 5.0,
  'reviewText': "I first read THE PROPHET in college back in the 60's. The book had a revival as
  did anything metaphysical in the turbulent 60's. It had a profound effect on me and became a
  book I always took with me. After graduation I joined the Peace Corps and during stressful
  training in country (Liberia) at times of illness and the night before I left, this book
  gave me great comfort. I read it before I married, just before and again after my children
  were born and again after two near fatal illnesses. I am always amazed that there is a
  chapter that reaches out to you, grabs you and offers both comfort and hope for the future.
  Gibran offers timeless insights and love with each word. I think that we as a nation should
```

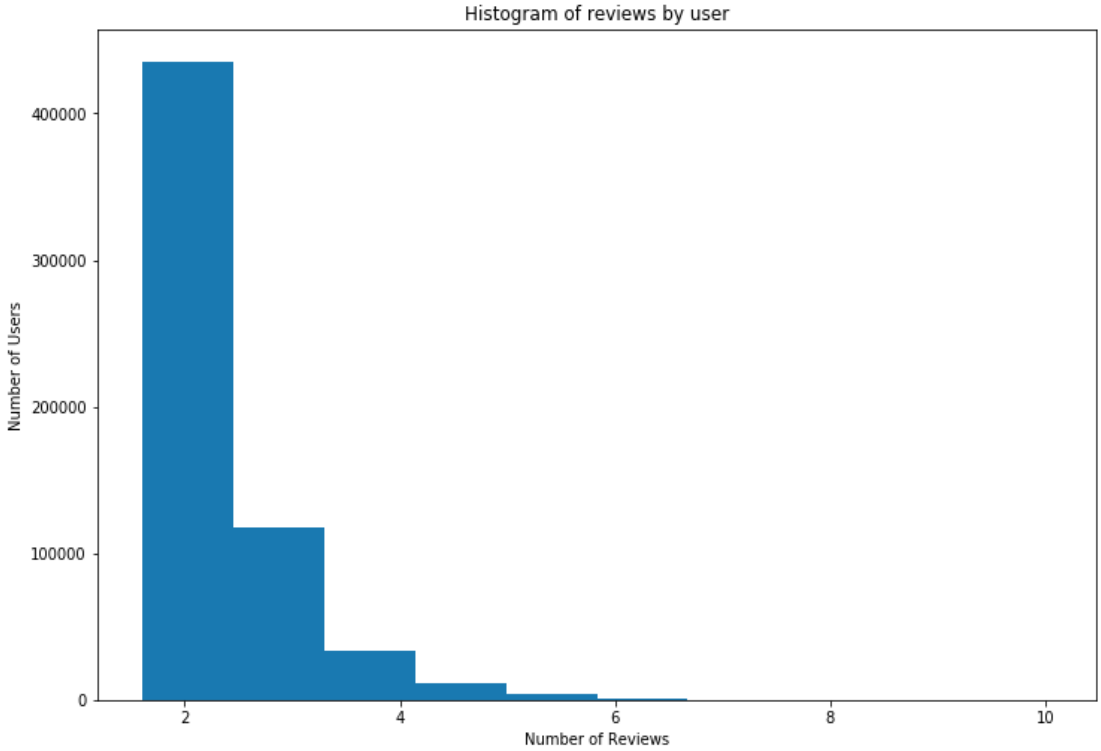
```
 read AND learn the lessons here. It is definitely a time for thought and reflection this
 book could guide us through.",
 'reviewTime': '09 27, 2011',
 'reviewerID': 'A1MOSTXNIO5MPJ',
 'reviewerName': 'Alan Krug',
 'summary': 'Timeless for every good and bad time in your life.',
 'unixReviewTime': 1317081600},
{'asin': '000100039X',
 'helpful': [7, 9],
 'overall': 5.0,
 'reviewText': 'A timeless classic.  It is a very demanding and assuming title, but Gibran
 backs it up with some excellent style and content.  If he had the means to publish it a
 century or two earlier, he could have inspired a new religion.From the mouth of an old man
 about to sail away to a far away destination, we hear the wisdom of life and all important
 aspects of it.  It is a messege.  A guide book.  A Sufi sermon. Much is put in perspective
 without any hint of a dogma.  There is much that hints at his birth place, Lebanon where
 many of the old prophets walked the Earth and where this book project first germinated most
 likely.Probably becuase it was written in English originally, the writing flows, it is
 pleasant to read, and the charcoal drawings of the author decorating the pages is a plus.
 I loved the cover.',
 'reviewTime': '10 7, 2002',
 'reviewerID': 'A2XQ5LZHTD4AFT',
 'reviewerName': 'Alaturka',
 'summary': 'A Modern Rumi',
 'unixReviewTime': 1033948800})
```


While the text, time, and helpfulness would likely be useful in a more complex recommendation algorithm, for the purpose of this project, I restrict the analysis to only the 1-5 rating given by the user. The resulting dataset used for analysis includes 8,898,041 ratings for 367,982 items from 603,668 users. Ratings range from 0-5, and averaged 4.25 with a standard deviation of 1.06. All ratings are integer values.



Clearly, most users are fairly generous with their ratings. So, without any big model to predict ratings, we can guess a user will rate an item they purchased a 5, and get it right more times than not. In fact, if we guess a user rated an item 5, we would be right 56% of the time, and within 1 point another 25% of the time. Additionally, if we just guess 4.25 for everyone, we'll only be off by an average of about 1.06, as shown by the standard deviation. These values give us somewhat of a baseline, although I refine the baseline estimate further in the following sections.

It's also important to note the sparsity of the ratings. For any given user item pair, it's unlikely there will be a review. In fact, the average number of reviews per user is 14.7, median 8, and mode 5. There are a few prolific users who have posted several thousand reviews, but for the most part, the vast majority of ratings will be unknown. The histogram below shows the distribution of number of ratings submitted by each user. Because of the range, a log scale is used.




## 2.1 Algorithms

Rather than focus on a single algorithm, this project explores 5 different algorithmic techniques with the same goal, on the same data. In the following sections, I describe the mathematical details of each, and how they will be implemented.

Recommendation systems occupy a unique space in machine learning. While they're probably closest to supervised learning, they are different from other supervised learning problems, in that the target is also the input feature. In a way, recommendation systems can be thought of as a combination of supervised and unsupervised learning. Supervised in that we are attempting to train an algorithm that minimizes the error against future ratings. But also unsupervised in that we are attempting to build a model that describes some underlying features of the data.

If we think of unsupervised learning as machine learning techniques used to describe some underlying characteristics of the data, we can equate this with building the model in the recommendation case. For example, there might be 20 underlying features that can be applied to some individual to predict how they will rate some item. If we knew these 20 features, the problem would be a simple supervised learning algorithm. Train the 20 features on the target, minimize the error, and attempt to make new predictions on future data.

In the recommendation system case, however, we don't know those 20 variables, we only know

the user's past ratings. So what we are essentially trying to do is use the ratings we know as a proxy for those unknown variables, creating a description of the data that tells us something about each users, and gives us some idea how they will rate future items. For this reason, we have to work to combine some supervised and unsupervised techniques.

### 2.1.1 Baseline

Before jumping into the individual algorithms, it's important to set a baseline of performance. Previously, we saw that the average rating across all the data was 4.25 with a standard deviation of 1.06. But maybe we can improve this. Perhaps certain users or items just tend to have higher ratings. Maybe some users are very generous, while other are very critical. For example, if one users keeps giving ratings of 1 and 2, we might think that same user will tend to give the same reviews for future items. Similarly, an item that keeps getting ratings of 2 or 3 would probably be expected to get similar ratings going forward.

Therefore, I set a baseline by splitting the data into training and test sets. On the training set, I get the overall mean rating for all items, as well as the group by means by items and users. I then join these means with the test set. For each user, I guess their mean from the training set as the rating they will give to future items. I do the same for each item. By chance, there can be users and items that appear in the test set but not the training set (there were 2 users and 5 items). In these cases, I guess the overall mean.

| Baseline Performance (RMSE) | |
|---|---|
| Overall Mean | 1.06 |
| Item Mean | 1.01 |
| User Mean | 0.97 |

Guessing the mean by item and user does improve performance. User mean does slightly better than item mean. The rest of the techniques follow this similar format, combining unsupervised and supervised methods to generate some feature for users and items based on the ratings, then using those features to predict future ratings.

The remainder of this section focuses only on the math and implementation of each algorithm. Results will be discussed in the subsequent section.

### 2.1.2 Linear Regression

Normally we would think of linear regression as a purely supervised learning technique, and that is largely how I use it in this case. Using the data from the baseline model, we can imagine a scenario where user and item means are correlated. Meaning users that tend to give high ratings tend to review items that also get high ratings (perhaps users who consistently research items before purchase are both purchasing higher quality items, and giving better reviews). Just using the mean of each doesn't account for such a correlation. And therefore might miss some potential inference. A simple regression model of

$$rating \sim b_1 \cdot user\_mean + b_2 \cdot item\_mean + intercept$$

might improve predictions on the test data. Additionally, we might think the effect of user or item ratings change at different levels of the other. For example, a high quality item might increase the likelihood of a generous user giving a good rating, but might not have the same effect for a more critical user (the opposite could also be true). We can account for such a possibility by including an interaction term that multiplies the user and item means together.

$$rating \sim b_1 \cdot user\_mean + b_2 \cdot item\_mean + b_3 \cdot user\_mean \cdot item\_mean + intercept$$

This gives a relatively simple starting model where we can effectively build the underlying features ourselves.

### 2.1.3 Collaborative Filtering

Collaborative filtering is used to describe a number of different algorithms when applied to recommendation systems, so I start by reiterating the basic concept. Essentially, the idea is to start with user and item means, as in the previous cases, but then correlated users and items with each other. The mean values are then adjusted when applied to future items. For example, we might find that two users tend to review similar items, and give similar ratings. Therefore, if we want to estimate how one user will rate an item that has been reviewed by the other, we can adjust our estimate based on the value given in the previous review. This also works if two users tend to give opposite reviews. We might adjust our estimate of the rating in the opposite direction, if we find that two users tend to give divergent reviews. I'll use this general technique in cosine similarity, singular value decomposition, and neural networks.

### 2.1.4 Cosine Similarity

Think of each user's ratings of all items as a vector, where each rating contributes to the magnitude and length of that vector. This vector will be huge in terms of number of possible dimensions (equal to the total number of items in the entire dataset), but will only have a few actual values. Now consider two users with two different vectors. If these users tend to have the same tastes, and have reviewed the same items, their vectors will likely point in similar directions. Even if one user is more generous than the other, if their reviews are similar when normalized, the vectors will be the same. Therefore, we can say that two users whose vectors point in the same direction are very similar, while two users whose vectors point in opposite directions are very different. Both of these cases can be useful in describing user similarity and new ratings, because any correlation between users will provide some predictive power on new ratings. If two users have only reviewed different items, or show no correlation in their reviews, the vectors will be orthogonal to each other, and will therefore provide no information about new ratings.

We can determine the degree to which two users are similar or dissimilar using the cosine similarity.

$$S_u^{cos}(u_k, u_a) = \frac{u_k \cdot u_a}{||u_k||||u_a||} = \frac{\Sigma x_{k,m} x_{a,m}}{\sqrt{\Sigma x_{k,m}^2 x_{a,m}^2}}$$

Where $u_k$ and $u_a$ are two users, and $x_{k,m}$ and $x_{a,m}$ are the ratings they give to some item m, summed across all items.

Therefore, the estimated rating for some new item for a user $k$ can be calculated as

$$\hat{x}_{k,m} = \bar{x}_k + \frac{\Sigma_{u_a} sim_u(u_k,u_a)(x_{a,m} - \bar{x}_{u_a})}{\Sigma_{u_a} |sim_u(u_k,u_a)|}$$

We start with the user mean $\bar{x}_k$, then sum across all users who have rated the same item, and have a similarity score to user k. We take a given user's rating on this item and subtract that user's mean rating across all items. Then multiply the result by the similarity score to user k. After summing across all users for which we have similarity scores, normalize by the number of users in the summation. Finally, use this value to adjust user k's mean for the new rating.

Some analysis of this algorithm reveals a glaring computational problem. While this works fine with datasets on the order of a few hundred or thousand users and items, the number of computations grows at rate $U \cdot I$ where $U$ is the number of users, and $I$ is the number of items. On this dataset, that means calculating the full user similarity will result in a matrix of size around 4 terabytes.

However, a bit more analysis also reveals that the vast majority of users will have not reviewed any items in common, and their similarity scores will then be zero. Further, since we're only interested in scores for items in the test set, we can restrict similarity score calculations to only those related to the items of interest. While still computationally difficult, the problem because much more manageable versus calculating the entire similarity matrix.

While scipy does include a cosine similarity function, it had to be modified in order to make these reduced calculations. The full description of the modification can be found in the accompanying code file, but the basic idea was to create a reduced user item sparse matrix with only the users and items relevant to predicting the a given user-item rating. This also meant running the method in a loop, which sacrifices some performance in terms of the SIMD capabilities of scipy, but overall reduced running times by several orders of magnitude (specific performance values are given in the results section).

#### 2.1.5 Singular Value Decomposition

The general idea of a singular value decomposition is to create a set of three matrices that, when multiplied, return the original matrix.

$$M_{(A,B)} = U_{(A,A)}\Sigma_{(A,B)}V^T_{(B,B)}$$

Where M is an A X B matrix. U and V are unitary orthogonal matrices, and $\Sigma$ is a diagonal matrix. This can be thought of as similar to principle component analysis. The $\Sigma$ matrix contains the eigenvalues from the original matrix, and $V$ contains the eigenvectors. U is then used to contain the information necessary to reconstruct the original matrix.

In the form describe above, the singular value decomposition can exactly reconstruct the original matrix by recording all eigenvalues of a matrix, and therefore producing two orthogonal matrices of similar size to the original matrix. What we're really interested in is the orthogonality of the new matrices. In the case of our data, these matrices can be thought of as containing latent features of the users and item that determine their ratings. So the rows of $U$ can be thought of as describing some unknown features of the users, while the rows of $V$ describe features of the items.

Additionally, if we rank the eigenvalues in $\Sigma$ from greatest to smallest, we get the most important user and item features leading each matrix. So instead of reconstructing the entire original matrix, we can come up with some set of features to approximate the original matrix. It's these features we can use to generate new predictions.

For example, say we want to generate twenty latent features to describe the original matrix, instead of building the fill equation above, we build a reduced set of matrices.

$$M_{(A,B)} \approx U_{(A,20)}\Sigma_{(20,20)}V^T_{(20,B)}$$

This gives us a smaller set of matrices that approximate the original matrix by taking its most important features as defined by its first 20 eigenvalues. We can then take the corresponding user and item rows in $U$ and $V$ for a given rating we wish to predict. We can then use the resulting value in a similar form to the cosine similarity used previously, to adjust the user mean and get a rating for some new item.

While reconstructing the entire matrix with the Amazon data would run into the same computational problems as cosine similarity, building a set of reduced matrices to approximate the original is much more computationally tractable. Luckily, because user and item scores will likely be significantly correlated, a relatively few number of eigenvalues can give a strong description of the original data.

Even more luckily, a new version of scipy includes a method for computing a singular value decomposition on a sparse matrix, making the described computations relatively easy and fast, even for very large amounts of data.

#### 2.1.6 Restricted Boltzmann Machine

A restricted Boltzmann machine is a form of neural network suggested by Paul Smolensky and later refined by Geoffrey Hinton, and is credited with the renewed interest in neural networks that

began in the mid 2000s. An RBM consists of two layers of neurons, visible and hidden. Data is fed into the visible layer, which then activates neurons in the hidden layer. These hidden neurons then try to reconstruct the original data. In some sense, this can be thought of as a data compression method, similar to SVD, where the hidden layer contains some latent features of the original data that it uses to approximate the original information.

More precisely, the RBM is a fully connected bipartite graph, in which hidden nodes are activated as a Bernoulli function of the input values. For a given set of visible units $v$ with biases $a$, hidden units $h$ with biases $b$, with weights $w$, we can calculate the energy of a configuration as

$$E_{(v,h)} = -\Sigma_i a_i v_i - \Sigma_j b_j h_j - \Sigma_i \Sigma_j v_i w_{i,j} h_j$$

Our goal is to find some configuration of weights and biases that minimize the system's energy. We do this with a training algorithm known as contrastive divergence. The steps of the algorithm are:

1) For a training sample v, compute a set of probabilities of the activation of the hidden layer as $v \cdot W$ where $W$ is a matrix of weights between v and h. Feed the resulting values into a sigmoid activation function to ensure they have values between 0 and 1.

2) Select a set of hidden nodes to activate by drawing from the probabilities output by the activation function.

3) Compute the outer product $(v \otimes h)$ for the visible and hidden nodes. This will become the positive gradient.

4) Reconstruct a sample of the visible units $(v')$ by feeding the hidden units as inputs to the weights. Resample the hidden units $(h')$ using the reconstruction.

5) Compute the outer product $(v' \otimes h')$ for the reconstructed visible and hidden nodes. This will become the negative gradient.

6) Update the weights by taking the difference in the positive and negative gradients times some learning rate.

7) Update the biases by uses the difference in the original and reconstructed values for each layer.

Contrasted divergence can be implemented similar to any other gradient descent algorithm. For this project, I used batch gradient descent, where a set of observations are trained for each update of the weights. Additionally, I make one modification for the recommender case. A mask is applied to the reconstructed data so only values for which there is a rating to train on are considered in the contrasted divergence. Because the vast majority of elements of each vector will be zero, without a mask the algorithm will learn to guess zero in every case. However, we want to zero cases to be predictions on some new data. Therefore, we only want to train on values for which we have data.

Inference is then performed by feeding in an existing vector from the training set for a given user. The vector is then reconstructed without the mask in order to get estimates of all the unknown values.

#### 2.1.7 Autoencoder

Compared to the RBM, autoencoders are more similar to common neural networks. Think of the case of a simple feedforward neural network, perhaps for the mnist data. The output will have 10 neurons, one for each category. The idea is then to feed in a set of pixels that activate the correct output neurons. Imagine if instead of the 10 categories, the output neurons were the same size as the input. The loss function could then be calculated as the difference between the input and the output. Then, using standard backpropagation, the network can be trained to reconstruct the input, rather than predict categories.

If we construct a simple multilayer feedfoward network that takes a vector of user ratings as input, and attempts to reconstruct that vector at output, it gives a method of predicting new ratings. Like with the RBM, we run into the same problem of the network predicting all zero because most of the observations are zero. So, like the RBM, we can apply a mask so the loss function and gradient descent are only performed as a function of the valid inputs. Any outputs for new values will be ignored during training.

Inference is then just a matter of feeding in an existing vector from the training data, and taking the output layer as the prediction of ratings for all items.

# 3 Data Preprocessing

As discussed in the previous section, the data is in the form of a large json file. Since we are only interested in the user and item ids, and ratings, these can be extracted to a more manageable dataset. To do this, I read the data using a Dask bag, convert the data elements to arrays, then combine them to a single Pandas dataframe. The advantage of using dask is that it avoids reading the entire json file into memory, as well as its ability to parallelize read operations.

The Pandas dataframe is then converted to all numeric values using SKLearn's label encoder class. This allows for easily converting the data to different formats while ensuring the same matching between users and items. At this point the data is stored in a long edgelist type format. An example is given below.

| | asin | rating | reviewerID |
|---|---|---|---|
| **0** | 0 | 5.0 | 551 |
| **1** | 0 | 5.0 | 284567 |
| **2** | 0 | 5.0 | 52489 |
| **3** | 0 | 5.0 | 101774 |
| **4** | 0 | 5.0 | 310083 |

At this point, the dataframe is split into training and test sets using SKLearn's train_test_split function. 10% of the dataframe was used for testing. A consistent random seed was also used, to ensure the same test data was used for comparison across algorithms.

This long format was used for both the baseline means, and the linear regression model. From the training data, groupby means were used to create predictors for the test data. For the remaining algorithms, the long format was not feasible. Instead, the data needed to be in the form of a sparse adjacency matrix. This matrix is of size $(number\_of\_users) \cdot (number\_of\_items)$. Because any given user is unlikely to have reviewed any given item, the vast majority of the entries will be zero. Because of the high degree of sparsity, representing the matrix in a dense form would be extremely inefficient. Instead, the matrix is constructed using scipy's sparse matrix functions. In the case of SVD, the algorithm used to compute the orthogonal matrices can take a sparse matrix as input. In the cosine similarity and neural network cases, a dense matrix is required. In these case, a small portion (100 rows) of the total matrix is converted for each batch of calculations. While this is computationally intense, it can be parallelized, and is much more tractable in terms of memory than dealing with the entire matrix.

# 4 Implementation and Refinement

For all algorithms, an array of predictions was generated, and compared to the actuals in the test set using root mean squared error. How exactly this was built varied by method. In the case of linear regression, the array was generated all at once by simply feeding the test set user and item arrays into a linear model trained on the training set. Missing values were then filled with the mean. In the case of SVD, once the orthogonal matrices are built, iterate over the arrays of user

and item ids, multiplying the respective rows of each matrix, storing the resulting values in a new array of predictions. For cosine similarity, once all the necessary similarity scores are computing, store them in a new dataframe. For each user item pair, find the corresponding similarity score and adjust the user mean, again storing the resulting value in an array of predictions. In both neural network cases, the algorithms output predictions for a given user on all items. Extract only items for which there is a corresponding user-item pair in the test set, and again add to an array of predictions.

For linear regression and cosine similarity, the features of the model are already predetermined, and so there are no hyperparameters to tune. However, SVD can be tuned in terms of the number of latent factors. Values between 10 and 50 were tried, and 20 found to maximize accuracy on the test set. The neural networks have even more tuning parameters. The RBM model can be tuned in terms of the number of hidden nodes. The autoencoder can be tuned in terms of both number of nodes and depth of the network. For both, node counts from 16 to 256 were tried, for the autoencoder depths from 1 to 5 were tried. 64 nodes worked best for both, while depth of 3 worked best for the autoencoder.

# 5 Results

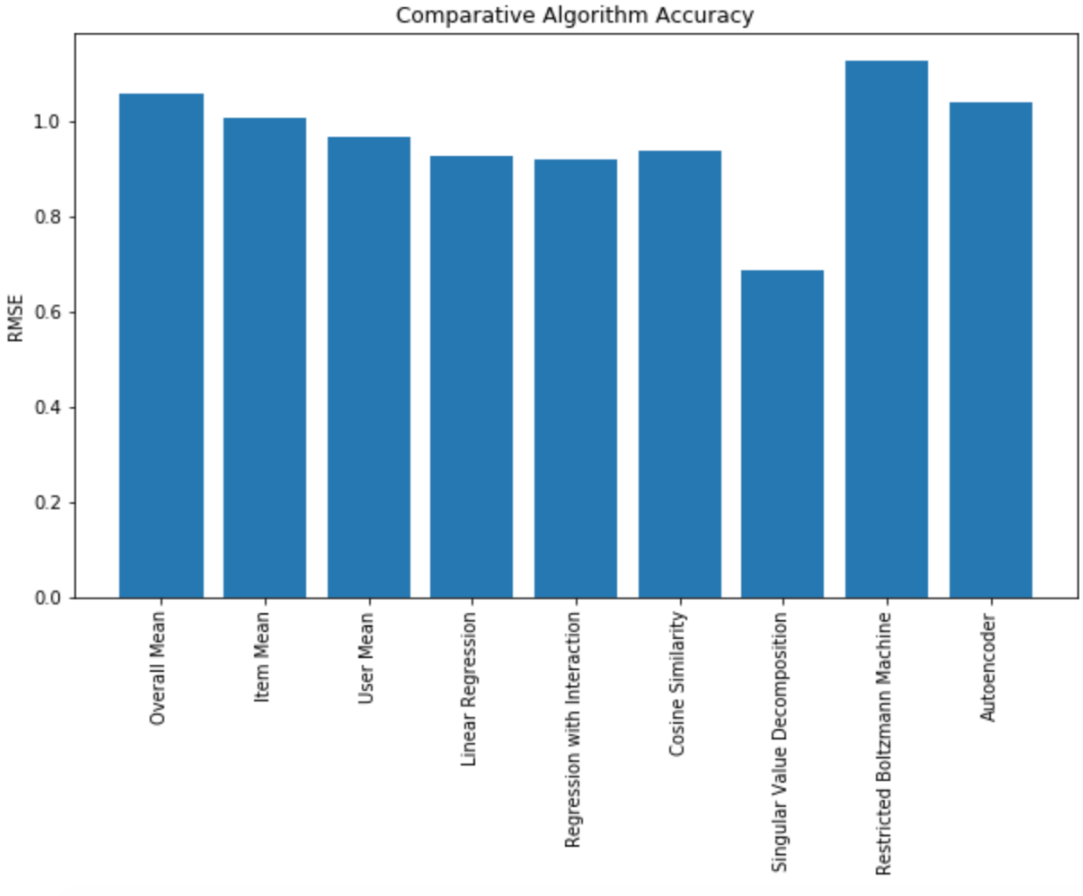
The accuracy and computational performance of all algorithms are listed below.

| Algorithm | RMSE | Computation Time |
|---|---|---|
| Overall Mean | 1.06 | 10 seconds |
| Item Mean | 1.01 | 10 seconds |
| User Mean | 0.97 | 10 seconds |
| Linear Regression | 0.93 | 30 seconds |
| Regression with Interaction | 0.92 | 30 seconds |
| Cosine Similarity | 0.94 | 8 hours |
| Singular Value Decomposition | 0.69 | 5 minutes |
| Restricted Boltzmann Machine | 1.13 | 3 hours |
| Autoencoder | 1.04 | 1 hour |

The different algorithms return a wide range of both accuracy and computational performance. The simple linear regression improves on the baseline, but not by a large amount. Linear regression does, however, offer the advantage of being computational efficient, only taking on the order of seconds to compute. Cosine similarity performs about as well as linear regression. This was surprising, as it is a common method for collaborative filtering. I suspect the data are too sparse in this case for the vectors used in cosine similarity to be meaningful. In additional, on a dataset this large, cosine similarity takes very long to calculate, around 8 hours. The neural networks perform much worse than expected. The RBM actually falls behind the baseline, while the autoencoder performs about as well as baseline. While this is disappointing, it's not all that surprising. Neural networks often require an enormous amount of tuning, both in terms of depth and nodes, but in terms of activation functions, and the structure of that data. It actually brings to mind a joke I recently overheard, of a data scientist discussing how his team of 18 people spent 4 months getting a neural network to outperform logistic regression. With further tuning, the neural networks can probably be improved, but in terms of off the shelf algorithms, which is the goal of this comparison, the significantly underperform simpler algorithms. Finally, singular value decomposition standouts by far in terms of accuracy. By a wide margin, it produces the most accurate predictions on the test dataset. In addition to its accuracy, it is one of the faster algorithms to calculate. This is primarily due to the fact that scipy already has highly optimized functions to compute the SVD matrices from a sparse matrix.

# 6 Conclusion

The plot below summarizes the results of all the tested algorithms.




Singular value decomposition is the clear winner in this case. It not only significantly outperforms everything else, but does so with relatively high computational performance. Neural networks have shown greater performance on some datasets. In this case, they might have suffered due to the high sparsity of the data. They also might need greater tuning in order to obtain their top possible performance.

A simple way to conclude is what I might recommend to a company looking to implement a recommendation system. If it were a company with large resources, and high quality data with less sparsity, I might recommend researching the neural network methods. In terms of a simple off the shelf algorithm that can be quickly and easily implemented, I would recommend using singular value decomposition.