Collaborative Filtering: Neighborhood and Latent Factor Models

GitHub Repository

Giulio Duregon
gjd9961

Joby George
jg6615

Jonah Poczobutt
jp6422

Abstract

Since the advent of digitized consumption of goods and services, and galvanized by the "Netflix Prize" (Töscher and Jahrer, 2009), Recommendation Systems have been researched deeply by the scientific community. In this report, we explore several implementations of modern day Collaborative Filtering models, and evaluate their performance against a baseline: a non-personalized, popularity-based, basket-of-goods model. The primary focus of this report will be PySpark's (Zaharia et al., 2016) implementation of an Alternating Least Squares, Matrix-Estimation Latent Factor model (Koren et al., 2009). Additionally, we will implement a single-machine ALS implementation using LensKit (Ekstrand, 2018), and a fast-search recommendation approach using Annoy. Training, validation, and test sets are derived from the MovieLens (Harper and Konstan, 2015) data set, and model performance is evaluated on the top 100 recommendations ordered descending by relevance. Evaluation metrics will measure predictive success of implicit and explicit user feedback, as well as model fitting and inference times.

1 Introduction

Since the advent of digitized consumption of goods and services, and galvanized by the "Netflix Prize" (Töscher and Jahrer, 2009), Recommendation Systems have been researched deeply by the scientific community. Recommendation Systems ultimately explore a classification challenge by taking in user and item data, and predicting user item feedback. User feedback is captured either implicitly or explicitly, with each feedback mechanism designed to address disparate questions:

1. Implicit Feedback - *"Will a user interact with the item we recommend?"*

2. Explicit Feedback - *"Will a user interact with an item we recommend in a favorable way?"*

Many different approaches exist to measure feedback and evaluate recommendation system performance, from non-personalized basket-of-goods models, clustering methods, and supervised-learning Collaborative Filtering algorithms. For user-personalized approaches, success is imperative on models learning explicit or implicit pairwise similarities between users and items.

In this report, we explore several implementations of modern day Collaborative Filtering models, and evaluate their performance against a baseline: a non-personalized, popularity-based, basket-of-goods model. The primary focus of this report will be PySpark's (Zaharia et al., 2016) implementation of an Alternating Least Squares, Matrix-Estimation Latent Factor model (Koren et al., 2009). Additionally, we explore trade-offs between a distributed approach (PySpark), and a single-machine ALS implementation using LensKit (Ekstrand, 2018). We also evaluate a fast-search approximate nearest neighbors approach using Annoy. Further discussion of model selection can be found in Section 4.

We will fit, predict, and evaluate our models using the MovieLens (Harper and Konstan, 2015) small and large data sets. Model evaluation will be measured by several metrics designed to capture explicit and implicit feedback mechanisms, which will be calculated using the top 100 recommended movies for each user, (sorted descending by predicted relevancy), against observations in the evaluation data sets. Further discussion of data set preprocessing can be found in Section 2.2, while model evaluation is outlined in Section 5.

One limitation inherent with our project is that we evaluate model performance on observational data, as we can never probe a user to see whether they truly would enjoy our recommended movie. To simulate a hypothetical in-market performance of our recommender system, while enforcing an unbiased model, we create train, validation, and test data splits that follow certain criteria:

1. The first 60% of each users reviews, (ordered ascending by date), are assigned to the training data set

2. The remaining 40% of each user reviews are

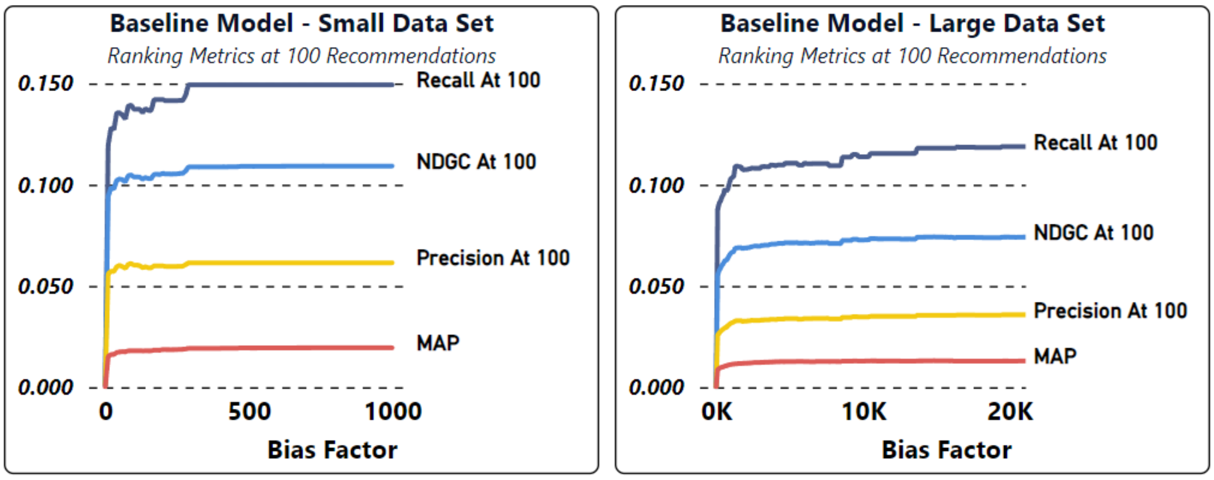

Figure 1: Baseline Model Performance on Small and Large Validation Set - Hyper Parameter Tuning

assigned to either the validation data set, or the testing data set

3. No users appear in both validation and testing data sets

Further discussion of train, validation, and test split methodology can be found in Section 2.3.

Ultimately, we aim to learn how to build, scale, and optimize recommender system models with a variety of modern day Collaborative Filtering methods.

2 Data & Preprocessing

2.1 MovieLens Movie Ratings Data Set

We used the MovieLens (Harper and Konstan, 2015) data sets provided by the GroupLens organization to fit and evaluate our Collaborative Filtering models. The MovieLens data set consists of 5-star ratings between movie-watchers (indexed by *userId*), and movie titles (indexed by *movieId*).

Two data sets were used and evaluated separately, *ml-latest-small*, which we will refer to as the "Small data set", and *ml-latest*, which we refer to as the "Large Data set". At the time of this report, both data sets were last updated in 2018.

Each data set consisted of several csv files. For our analysis of both data sets, the files of interest were the *movies.csv* and *ratings.csv* files, which mapped to one another on *movieId* to create a sparse Utility Matrix.

Joining the data sets on *movieId*, we were able to analyze our resulting Utility Matrices. The Small data set has 100,830 ratings, 610 distinct *usersId*s, and 9,724 distinct *movieId*s, meaning only 1.67% of entries in the corresponding user-item Utility Matrix have values. For the Large data set, there were 27,753,444 ratings, 283,228 distinct user IDs, 58,098 distinct movie Ids, meaning only 0.018% of entries in the corresponding Utility Matrix had values.

2.2 Preprocessing

2.2.1 Data Cleaning

Initial data exploration quickly revealed that the *title* and *movieId* fields did not have a 1:1 mapping. Alarmingly, we noted that in the MovieLens (Harper and Konstan, 2015) data set there were distinct *userid*s which reported multiple ratings for the same movie *title* mapping to multiple distinct *movieId*s. Concerned about the validity of these data-points, we decided to de-duplicate our data and enforce a 1:1 mapping between *title* and *movieId*. We discarded duplicate *movieId*s that mapped to the same *title* by discarding the *movieId* with the least ratings, and including only the *movieId* with the most rating observations.

2.2.2 Transformations

For our Alternating Least Squares (ALS) (Koren et al., 2009) model implementation, we will follow the third Normalization procedure outlined by (Leskovec et al., 2014). The technique is as follows: for every non-blank rating $M_{i,j}$ we will subtract one half the sum of the mean rating across all non-blank movies of the i^{th} user, and the mean rating across all non-blank users for the j^{th} movie.

The rating normalization procedure results in an increased cosine distance for dissimilar users and for dissimilar movies. The new representation of movie rating aids the ALS model during the

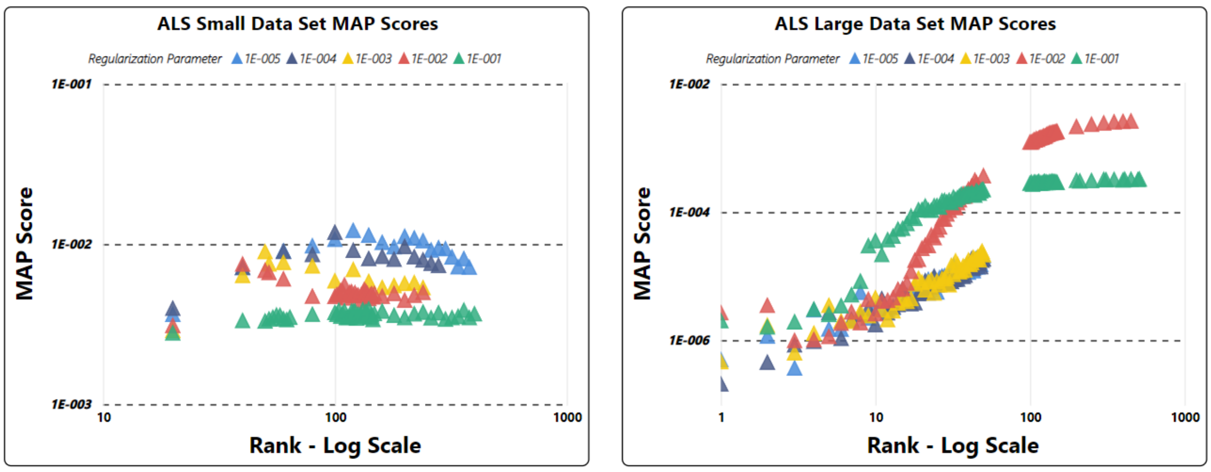

Figure 2: ALS Hyper Parameter Tuning - $\mathit{rank} \in [1, 500]$, $\mathit{regParam} \in [10^{-5}, 10^{-1}]$, $\mathit{maxIter} = 10$

$$M_{i,j} = M_{i,j} - \frac{avg(M_i) + avg(M_j)}{2} \qquad (1)$$

Equation 1: ALS Utility Matrix Normalization

Matrix Decomposition stage, helping it discover the Latent Factors that compose the Utility Matrix, and thus discover user and movie archetypes that define their similarities and differences. To ensure fair model evaluation, once predictions have been generated the normalization process is reversed, and the respective user and item means are added back to our estimated elements on the utility matrix.

2.3 Train / Val / Test Splitting

For accurate evaluation, creating validation and test data sets that closely mimicked the potential real-world implementation of our models is imperative. To accomplish this, we needed to make sure that sufficient data for every user appeared in our training data sets, and that no data leakage occurred between validation and testing data sets.

To implement the above criteria, our strategy was to take the first 60% of a user's reviews, sorting by date, and assigning these data points to our training data set. To ensure there was no data leakage from the validation to the test sets, we created disjoint evaluation data sets based on *userId*. Therefore, no single user would have ratings in both the validation and test set. Furthermore, we held an approximately equal split in our validation and test data sets, which each set containing around 20% of the remaining reviews.

By having disjoint *userId*'s we ensure that hyper-parameter tuning optimization on the validation data set does not influence predictions on our test set, as they are independent in regards to users. If users were present in both the validation and test data sets, our test error might not be reflective of performance on truly unseen data. This is because we may be over-fitting our hyper-parameters to the specific data points in our tuning process, rather than creating generalized latent factors.

To ensure reproducibility, upon creation the train, validation, and test splits for both the small and large data sets were saved on NYU's HPC Hadoop (Apache Software Foundation) file system.

3 Tools and Libraries Used

3.1 PySpark

Working with large data sets, we knew that computation parallelization would be necessary to reduce model run-time and to build efficient data pipelines. To accomplish this task, we used PySpark 3.0 (Zaharia et al., 2016) for our distributed computing needs. This allowed us to run distribute our computation over NYU's HPC cluster, and reduce the run time of our experiments greatly. Additionally, we leveraged PySpark's model implementation for our ALS task, which uses the results from *Matrix Factorization Techniques for Recommender Systems* (Koren et al., 2009)

3.2 LensKit

To investigate trade-offs between distributed and single machine approaches, we leverage the LensKit (Ekstrand, 2018) API to create a single-machine ALS model. Our LensKit ALS model is trained and evaluated on the same training, validation, and test splits created for our PySpark ALS

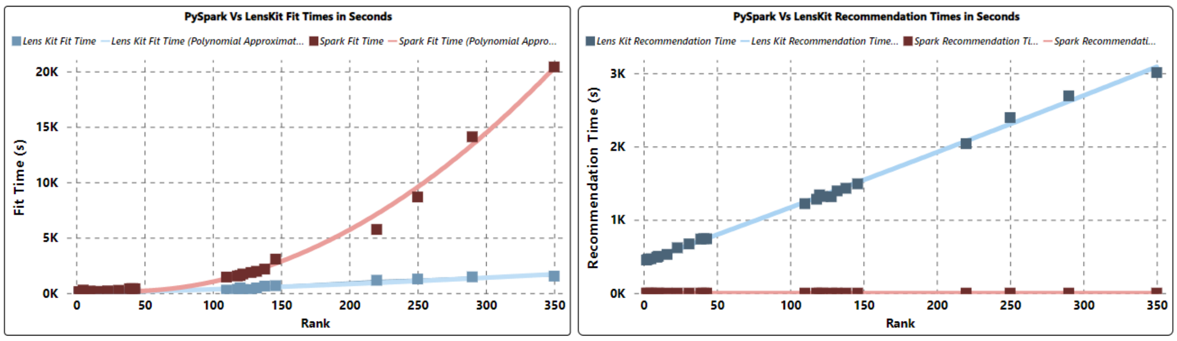

Figure 3: PySpark vs ALS Speed Comparison - $rank \in [1, 350]$, $regParam = 1.0 \times 10^{-2}$, $maxIter = 10$

model. Model fitting time, prediction time, and evaluation can be found in Section 6.

3.3 Annoy

For our fast search extension we used Spotify's Annoy - a tree based approximate nearest neighbor search. The Annoy algorithm takes in item latent factor representations of a Utility Matrix as input, and creates a binary tree to cluster similar items together. Each binary tree node bifurcates item latent factor vectors by taking their dot product with a random vector in $\mathbb{R}^d$, where d is the rank of the item latent factors. User latent factors can then be used to query the tree and retrieve movie recommendations. The latent factors used in our Annoy implementation were derived from the PySpark ALS model that performed best on the validation data set. Model fitting time, prediction time, and evaluation can be found in Section 6.

3.4 Visuals

To create visuals to illustrate our findings we used Microsoft's Power BI (Pow).

4 Model Selection and Tuning

4.1 Baseline Model

Our baseline model is a non-personalized, popularity-based, basket of goods model that recommends the same 100 movies to all users. Recommendations were ordered by relevance in decreasing order. Movie relevance was computed by taking the average rating of each movie.

One flaw with popularity based approaches is that highly rated movies with few interactions results in unstable metrics of relevance. The likelihood of these movies being relevant if included in our recommendation set is small. Increasing the robustness of our relevance metric, we added a bias term, β to the denominator of our calculation as seen in Equation 2.

$$Rel(j) = \frac{\sum_{i=1}^{n} M_{ij} \times \mathbb{1}[M_{ij} \in \mathbb{R}]}{\beta + \sum_{i=1}^{n} \mathbb{1}[M_{ij} \in \mathbb{R}]} \quad (2)$$

Equation 2: Baseline Movie Relevance Calculation

Where $M \in \mathbb{R}^{n \times m}$ is the utility matrix, the j^{th} column represents a movie j's reviews across all users, and $\mathbb{1}$ is an indicator function that evaluates to 1 if the entry at M_{ij} has a rating and 0 otherwise. For our baseline model, the β term acted as a dampening factor of movie relevance, weeding out outliers and increasing the likelihood of relevant movies being included in our recommendation set.

Tuning the baseline models on our validation data set, we iterated over different value of β, evaluating 4 different ranking metrics seen in Figure 1. Using the elbow rule, we found that $\beta_{small} = 300$ provided the best results for the small data set, while $\beta_{large} = 10,000$ provided the best results for the large data set across all 4 metrics. Specifically, for the best performing baseline model for the small data set had a MAP of 1.9×10^{-2}, and MAP of 1.3×10^{-2} for the large data set. Model performance on the test set will be discussed in Section 6.

4.2 Alternating Least Squares Model

For our first personalized model we selected PySpark's (Zaharia et al., 2016) implementation of the Alternating Least Squares (ALS) algorithm (Koren et al., 2009). ALS is a latent factor model, which takes in as input a sparse utility matrix, M, and through an iterative approach returns an estimation of M as the product of two lower-dimensional matrices, U and V. ALS loss is minimized via Gradient Descent with respect to RMSE, which represents the average squared difference between

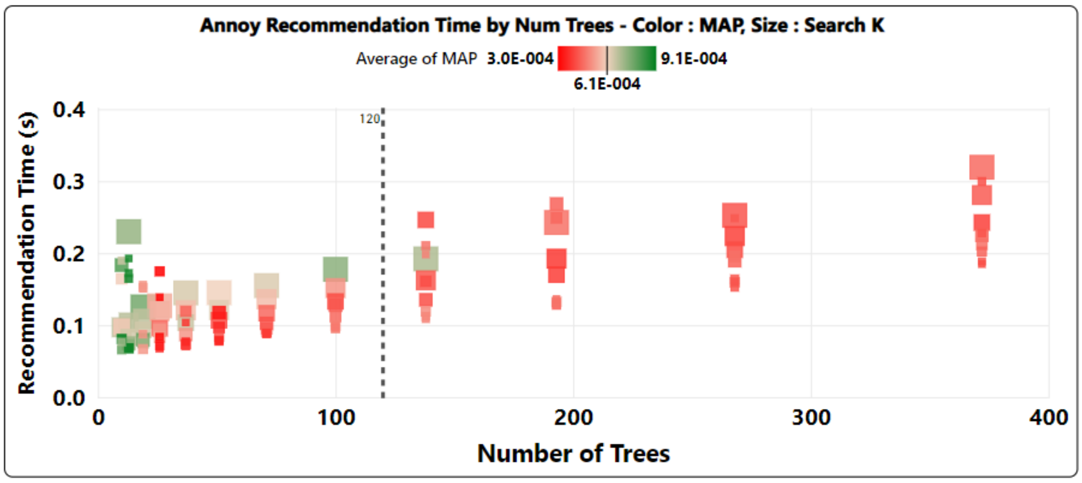

Figure 4: Annoy Results on Small Data Set - Color : MAP - Size : *search_k*

non-blank entries of M and the corresponding estimated entries of the matrix product UV. U and V each represent user and movie latent factors respectively, where each latent factor is a vector representation of abstract user or movie archetypes.

The key hyper-parameters for ALS are "*rank*" - the rank of our low dimensional Latent Factor matrices U and V, "*maxIter*" - which defines the amount of optimization iterations the ALS algorithm performs, and "*regParam*" - the λ regularization penalty. For reproducibility of results we set the "*seed*" parameter of our ALS models equal to 10.

We tuned our model by iterating over different parameters values for *rank* and *regParam*, while keeping max *maxIter* set to 10 to save computational cost and speed up our hyper-parameter search. As we primarily interested in answering questions regarding implicit feedback, we evaluated model performance on their respective MAP scores on the validation set, rather than the models RMSE performance.

Results on the validation data set - seen in Figure 2 - showed that the when the ALS model is run on the small data set it is not very sensitive to changes in *rank* and *regParam*, outputting similar MAP values across different combinations of parameters. However, when running ALS on the large data set, the results significantly different between varying parameters.

Using the elbow method for gauging our results, we concluded that for the small data set the best performing model on validation data had hyper parameters *rank* $= 120$, *regParam* $= 10^{-5}$ with $MAP = 1.2 \times 10^{-2}$, while for the large data set the best performing model had hyper-parameters *rank* $= 250$, *regParam* $= 10^{-2}$, with $MAP = 2.3 \times 10^{-3}$. The best performing model's parameters were selected to be when evaluating performance on the test set which can be found in Section 6.

4.3 ALS Single-Machine Implementation

As part of our evaluation of PySpark's ALS implementation, we compared our results to a single machine implementation of ALS using the LensKit (Ekstrand, 2018) package. Such a comparison makes it easy to evaluate any efficiency and, potentially, accuracy discrepancies resulting from our distributed computation approach (PySpark). We highlight train and predict times on the large data set, as a function of model rank, in Figure 3, fitted with degree two polynomial curves to visualize the relationship between model rank and run time. We find that the LensKit implementation fits faster but predicts slower than the PySpark implementation.

This result aligns well with our understanding of the ALS modeling process. In order to iteratively train our model, we need the results of the previous iteration to fit and change parameters, which limits opportunity for parallelization. However, as predictions are computed by taking the linear combinations of latent factor vectors, every user's prediction can be computed separately, thus the efficiency gains of PySpark's distributed computing are large. Accuracy comparisons between Spark and LensKit, on the validation set can be found in the Appendix in Section 7.2.

4.4 Approximate Nearest Neighbors Search

For our approximate nearest neighbor search we used Spotify's Annoy. The Annoy algorithm takes in item latent factor representations of a Utility Matrix as input, and creates a binary tree to cluster similar items together. Each binary tree node bifurcates item latent factor vectors by taking their dot product with a random vector in $\mathbb{R}^d$, where d is the rank of the item latent factors. User latent factors can then be used to query the tree and retrieve movie recommendations.

The algorithm has two key hyper-parameters: *n_trees* - the number of binary tree splits it creates, and *search_k* the amount of nodes to search for nearest neighbors during a query. Varying the two parameters results in a trade-off between precision and query speed. Hyper-parameter tuning can be seen in Figure **insert figure**, while query run time is discussed in section 6.

For our approximate nearest neighbor search we used Spotify's Annoy. The Annoy algorithm takes in item latent factor representations of a Utility Matrix as input, and creates a binary tree to cluster similar items together. Each binary tree node bifurcates item latent factor vectors by taking their dot product with a random vector in $\mathbb{R}^d$, where d is the rank of the item latent factors. User latent factors can then be used to query the tree and retrieve movie recommendations.

The algorithm has two key hyper-parameters: *n_trees* - the number of binary tree splits it creates, and *search_k* the amount of nodes to search for nearest neighbors during a query. Varying the two parameters results in a trade-off between precision and query speed. After running experiments on the small data set, we noted that the best performance across query time and MAP came from limiting *n_trees* to less than or equal to the rank of the latent factors. Surprisingly, the best MAP scores were in queries that had *search_k* less than 100, which we found counter intuitive as we expected a positive relationship between the number of nodes searched and any precision metric. To view visuals illustrating this trade off, please refer to Section 7.2. Query run time is discussed in section 6.

5 Evaluation Metrics

Inherent to offline recommendation system modeling is the challenge of evaluation. Limited by purely observational data, we can never truly probe an end user of our product and understand the effectiveness of our recommendations. Furthermore, because our models are evaluated on data limited to the reviews in our validation and test data sets, we have no method of guaranteeing that users have seen the movies that we recommend, which thwarts our attempts to measure explicit feedback. To capture as much inference as we can regarding our models success, we measured a variety of metrics that capture both implicit and explicit user feedback. Additionally, model fitting and inference time will be measured to gauge the feasibility of model deployment to production.

To measure implicit feedback, we use PySpark's (Zaharia et al., 2016) built in ranking metrics, which are evaluated on the top 100 movie recommendations for each user sorted in descending order by relevance. For more detail on how the implicit feedback metrics are mathematically defined, refer to the PySpark Evaluation Metrics documentation.

To measure explicit feedback, we create two custom metrics, *Average Precision at Intersection* in Section 5.2.1, and *Average Recall at Intersection* in Section 5.3. We create these metrics by redefining the set of relevant documents for each user as the intersection between predicted movies, and movies that actually appeared in that users evaluation data set. *Average Precision at Intersection* allows us to examine the quality of our recommendations where the corresponding validation data exists. Likewise, we can measure where our recommendations fail at predicting explicit feedback, with *Average Recall at Intersection.*

5.1 Implicit Feedback Metrics

5.1.1 Mean Average Precision

Mean Average Precision (MAP) is our primary metric for implicit feedback, which we implement by leveraging the PySpark API. Put succinctly, MAP is a measure of prediction accuracy which incorporates the rank ordering of our top 100 movie predictions. We place high importance on this metric as it quantifies our belief that if a model ranks one prediction higher than another, and then the user does not interact with that prediction, then the model should be penalized proportionally to its confidence in its failed prediction.

5.1.2 Precision at K

Precision at K measures the precision amongst the first k documents recommended for a user against their evaluation data. In this metric, rank order of predictions is not taken into account.

5.1.3 NDGC at K

NDGC at K measures the precision amongst the first k documents recommended for a user actually appear in their validation data, much like *Precision at K*. However, for NDGC at K, rank order of predictions is taken into account.

5.1.4 Recall at K

Recall at K measures the recall amongst the first k documents recommended for a user actually appear in their validation data. In this metric, rank order of predictions is not taken into account.

5.2 Explicit Feedback Metrics

5.2.1 Average Precision at Intersection

To evaluate our models explicit feedback predictions on recommended movies, we created a metric applicable to all models mentioned in this report, which we named *Average Precision at Intersection* (API). The metric was designed to address an explicit feedback question: *"We think you would watch this movie and enjoy it - where we right?"*.

API measures precision for the intersection of movies j that in a user i's recommended movies and user i's rated movies in their evaluation data set. For the purposes of this metric, we define the set of retrieved documents, V, as our hold-out evaluation data. We also define R as our set of relevant documents, the top 100 recommended movies (personalized or not), for all n users in V.

As the ratings in the data set are in a 0-5 scale, we assume a positive rating as a rating above 2.5. We then define a set the set of positive evaluation ratings as: $V^+ = \{V_{i,j} \in V | V_{i,j} \geq 2.5\}$. Average Precision at Intersection across all users in an evaluation data set can then be calculated in the following way:

$$API = \frac{1}{n} \times \sum_{i=1}^{n} \frac{\left|R_i \cap V_i^+\right|}{|R_i \cap V_i|} \qquad (3)$$

Equation 3: Average Precision at Intersection

The above calculation allows us to determine the accuracy of truly relevant[1] predictions for any given user. As such, this metric is entirely geared toward measuring explicit feedback. We do not include any penalty for recommending movies that the user has not seen. While comparing this metric between users could be dangerous (given the size of the intersection can vary widely), we believe that evaluating different models on how they perform on this metric across a large number of users is a fair comparison, as long as one keep in mind exactly what is being measured.

[1]Truly relevant in this context means that the movie j we predicted for user i, appeared in their evaluation data set.

5.3 Average Recall at Intersection

Similarly to how API was calculated, *Average Recall at Intersection* (ARI) measures the average recall the intersection of relevant documents and retrieved documents. Using the set definitions for V, R, and V^+ defined above, ARI can be calculated in the following way:

$$ARI = \frac{1}{n} \times \sum_{i=1}^{n} \frac{\left|R_i \cap V_i^+\right|}{\left|V_i^+\right|} \qquad (4)$$

Equation 4: Average Recall at Intersection

5.4 RMSE

RMSE was calculated for the PySpark ALS model and LensKit ALS model. RMSE measures the squared distance between non-blank entries in the Utility Matrix, against the predictions generated by the matrix product of the U and V latent factor matrices learned during ALS. A model that predicts explicit feedback well will minimize RMSE.

6 Model Results

Using the optimal hyper-parameters outlined in Section 4, we evaluated our best performing models on the test evaluation data set, summarized above by Table 1. Optimizing for time, we set the Annoy *n_trees* equal to the rank of each ALS model with respect to model size, and *search_k* to 100. To ensure model convergence, we set *maxIter* to 25 for all ALS models.

We found that the baseline model performed much better than we had anticipated, beating all other models with respect to measuring implicit feedback. We believe this could be due to the positively skewed data set, as 87% of user-item ratings in the large data set had a movie rating greater or equal than 2.5. More research is needed to determine model type performance differentials on a

		Model Fitting Time, Prediction Time, and Evaluation Metrics					
		Fit / Pred (s)	MAP	Prec at 100	API	ARI	RMSE
Small	Baseline	.251 / 0	1.6×10^{-2}	4.8×10^{-2}	9.2×10^{-1}	1.5×10^{-1}	–
	PySpark ALS	83.6 / .37	1.08×10^{-2}	3.0×10^{-2}	8.8×10^{-1}	1.1×10^{-1}	1.44×10^{0}
	LensKit ALS	38.2 / 6.97	1.27×10^{-2}	4.6×10^{-2}.	9.2×10^{-1}	1.0×10^{-1}	2.03×10^{0}
	Annoy	.46 / .08	9.1×10^{-4}	1.0×10^{-2}	8.7×10^{-1}	1.0×10^{-2}	–
Large	Baseline	.26 / 0	1.2×10^{-2}	3.4×10^{-2}	9.3×10^{-1}	1.22×10^{-2}	–
	PySpark ALS	2038.4 / .35	2.3×10^{-3}	8.3×10^{-3}	9.5×10^{-1}	2.6×10^{-2}	8.8×10^{-1}
	LensKit ALS	1263.9 / 2283.1	2.41×10^{-3}	8.82×10^{-3}	9.7×10^{-1}	2.7×10^{-2}	8.9×10^{-1}
	Annoy	8.56 / 3.12	1.5×10^{-4}	4.6×10^{-4}	8.2×10^{-1}	1.1×10^{-3}	–

Table 1: Model Evaluation Results on Small & Large Test Data Sets

less skewed data set. PySpark and LensKit's respective implementations performed remarkably similarly on evaluation metrics, with a slight edge to LensKit's model. As seen in Section 4, LensKit's fitting time was substantially lower than PySpark's, however its prediction time was magnitudes higher.

Lastly, our Annoy fast search was significantly faster at prediction time than its ALS counterparts, though was magnitudes worse at predicting both explicit and implicit feedback. Puzzlingly, we found that Annoy ran slower at prediction time than PySpark on the large data set. We theorize that we may have inaccurately calculated prediction times for the PySpark model, as we calculate the time difference before and after invoking a function call. PySpark leverages delayed computation, potentially rendering our approach ineffective. All models resulted in similar API, highlighting the need for additional metrics in order to evaluate the consequences of model/software selection.

7 Collaboration Statement

Jonah wrote the source code used for the baseline implementation model, the unit tests, and the API and ARI evaluation metrics.

Giulio wrote and maintained source code implementing PySpark's ALS model, and Annoy's tree-based fast search. He also ran the experiments on the NYU HPC cluster, constructed data pipelines, and created the Power BI visualizations seen in this report.

Joby wrote data-cleaning source code, and helped with the debugging by pulling log reports of locally compiled code to cross-validate with HPC results. Giulio and Joby collaborated on writing the source code responsible for generating Train/Validation/Test splits, while Jonah and Joby collaborated on the LensKit ALS single-machine implementation.

7.1 Software Installation Documentation

Installation instructions provided by LensKit, Annoy, and Power BI to utilize the software used in the report.

7.2 Additional Visuals

Some visuals were omitted from this report in the interest of space. More visuals can be found on the GitHub repository for this project, in the Plots folder.

References

Microsoft power bi. *Downloads | Microsoft Power BI.*

Apache Software Foundation. Hadoop.

Michael D. Ekstrand. 2018. The LKPY package for recommender systems experiments: Next-generation tools and lessons learned from the lenskit project. *CoRR*, abs/1809.03125.

F. Maxwell Harper and Joseph A. Konstan. 2015. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4).

Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37.

Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. 2014. *Mining of Massive Datasets*, second edition. Cambridge University Press.

Andreas Töscher and Michael Jahrer. 2009. The bigchaos solution to the netflix grand prize.

Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65.