E4571 Personalization

Final Project Report

Recommendation Systems Using Factorization Machine and Wide and Deep Learning

Yu Han Huang (yh3093), Deepak Ravishankar (dr2998)

1 - Objective

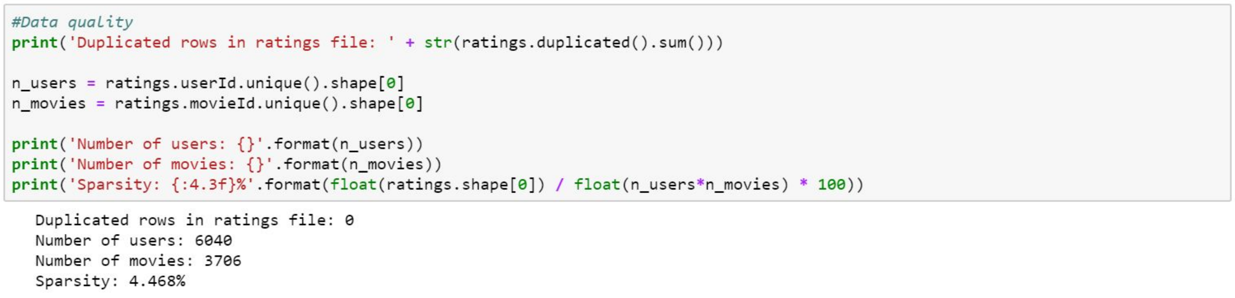
This project trained and evaluated two recommendation systems with two personalization approaches: Factorization Machine and Wide and Deep Learning. We used data from MovieLens as it is very comprehensive in the rating and user/item data that we would need in constructing such recommendation systems. The MovieLens datasets are also widely used in education, research, and industry, which remain one of the most long-standing and live research platform today.

In both systems, we want to learn how the algorithms can effectively recommend our users to the movies that they might like. We try to optimize our recommendation system to minimize the difference in our prediction of users' rating and their real preference. Since the dataset we use is relatively large, we are also interested in knowing how well these algorithms scale with the data size. More detailed objective and evaluation of individual system will be explained in the later part of the report.

2 - Data

We used the MovieLens 1M Dataset due to the following considerations:
1) It provides demographic information of the users and movie information which enables us to construct a factorization machine incorporated with additional information that can assist in prediction and can be compared with our original factorization machine.
2) The data size is sufficiently large with 1,000,209 anonymous ratings of 3,706 movies made by 6,040 MovieLens users who joined MovieLens in 2000.

Data Quality



```
#Data quality
print('Duplicated rows in ratings file: ' + str(ratings.duplicated().sum()))

n_users = ratings.userId.unique().shape[0]
n_movies = ratings.movieId.unique().shape[0]

print('Number of users: {}'.format(n_users))
print('Number of movies: {}'.format(n_movies))
print('Sparsity: {:4.3f}%'.format(float(ratings.shape[0]) / float(n_users*n_movies) * 100))
```

```
Duplicated rows in ratings file: 0
Number of users: 6040
Number of movies: 3706
Sparsity: 4.468%
```

The rating dataset is quite clean with no duplicated rows. The data is comprised of 3,706 movies made by 6,040 MovieLens users, and the sparsity is approximately 4.5%.

3 - Factorization Machine Recommendation System

Factorization Machine is a model class that combines the advantages of Support Vector Machines (SVM) with factorization models. Like SVMs, FMs are a general predictor working with any real valued feature vector. In contrast to SVMs,FMs allow for interaction terms for items within a single entity type. Thus, they are able to estimate interactions even in problems with huge sparsity where SVMs fail.

3.1 Objective

We built two types of factorization machine, one with and one without user and item features of side information. Our objective is to compare the two models to evaluate whether incorporating side information would improve the accuracy of Factorization Machine models. While experience has shown these models to suffer from slow training and local minima, we are also interested in knowing the model's performance in running time and accuracy when we subsample datasets from small to large and users/items from sparsely-populated to well-populated.

The accuracy metrics we used in evaluating the recommendation systems are Root Mean Square Error (RMSE) and Mean Absolute Error (MAE) . The two methods are the two of the most common metrics used to measure accuracy for continuous variables. We further devised an item-coverage method to test for quality beyond just accuracy metrics.

We also built a collaborative filtering recommendation system using k-Nearest Neighbors (kNN) as our baseline model. We will compare the two models to the baseline model in addition to determine whether it would be more robust to predict with latent factors than user/item similarities.

3.2 First model - Factorization Machine with no side information

The first model is a factorization machine with no side information, and are only user and item ratings.

3.2.1 Data Preprocessing

The data preprocessing consists of three parts - subsetting data, splitting test and train data, and one hot encoding data.

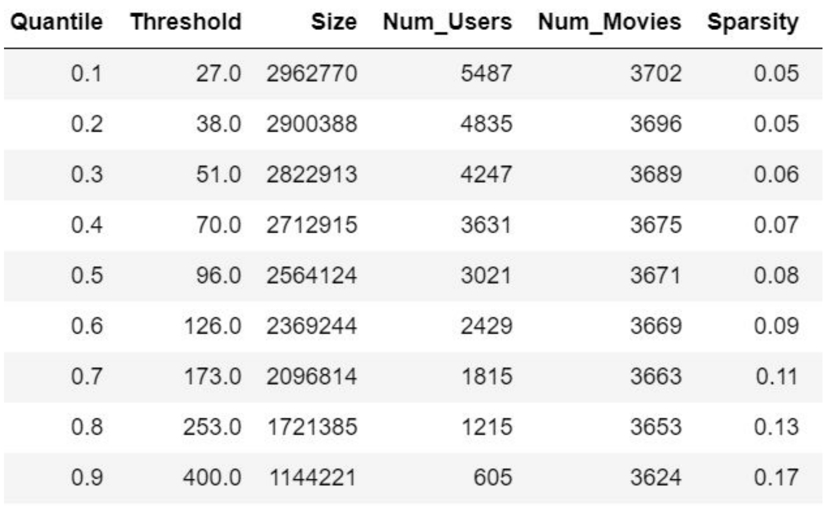
3.2.1.1 Subsetting Data

To subsample datasets from small to large, and subsample users/items from sparsely-populated to well-populated, we will subsample data by the following three methods:
a) Subset data from less prolific users to prolific users
b) Subset data from less popular items to popular items
c) Subset data in both user and item directions

We will compare the performance of different subsetting methods to evaluate which subsetting method works the best for factorization machines.

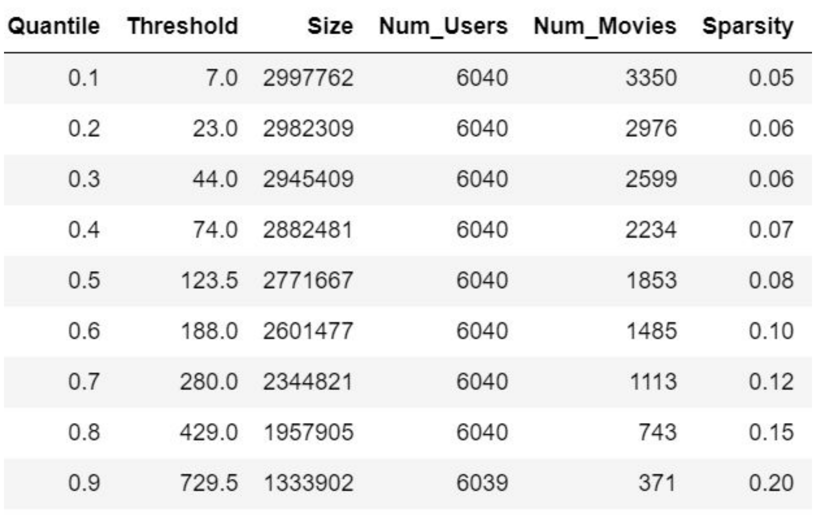
Sample size:

a) Subset data from less prolific users to prolific users



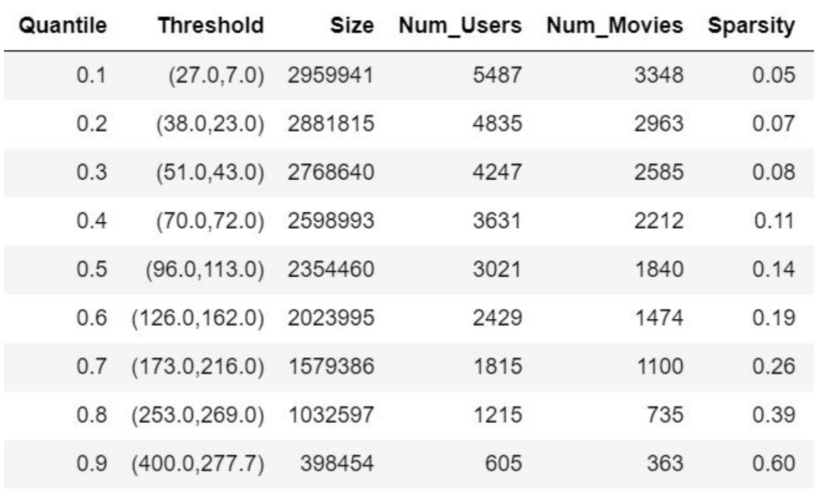
Quantile	Threshold	Size	Num_Users	Num_Movies	Sparsity
0.1	27.0	2962770	5487	3702	0.05
0.2	38.0	2900388	4835	3696	0.05
0.3	51.0	2822913	4247	3689	0.06
0.4	70.0	2712915	3631	3675	0.07
0.5	96.0	2564124	3021	3671	0.08
0.6	126.0	2369244	2429	3669	0.09
0.7	173.0	2096814	1815	3663	0.11
0.8	253.0	1721385	1215	3653	0.13
0.9	400.0	1144221	605	3624	0.17

b) Subset data from less popular items to popular items



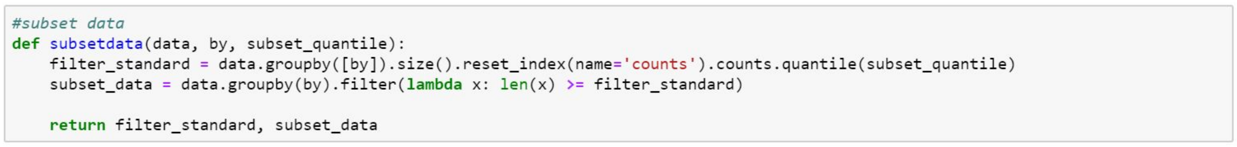
Quantile	Threshold	Size	Num_Users	Num_Movies	Sparsity
0.1	7.0	2997762	6040	3350	0.05
0.2	23.0	2982309	6040	2976	0.06
0.3	44.0	2945409	6040	2599	0.06
0.4	74.0	2882481	6040	2234	0.07
0.5	123.5	2771667	6040	1853	0.08
0.6	188.0	2601477	6040	1485	0.10
0.7	280.0	2344821	6040	1113	0.12
0.8	429.0	1957905	6040	743	0.15
0.9	729.5	1333902	6039	371	0.20

c) Subset data in both user and item directions



Quantile	Threshold	Size	Num_Users	Num_Movies	Sparsity
0.1	(27.0,7.0)	2959941	5487	3348	0.05
0.2	(38.0,23.0)	2881815	4835	2963	0.07
0.3	(51.0,43.0)	2768640	4247	2585	0.08
0.4	(70.0,72.0)	2598993	3631	2212	0.11
0.5	(96.0,113.0)	2354460	3021	1840	0.14
0.6	(126.0,162.0)	2023995	2429	1474	0.19
0.7	(173.0,216.0)	1579386	1815	1100	0.26
0.8	(253.0,269.0)	1032597	1215	735	0.39
0.9	(400.0,277.7)	398454	605	363	0.60

Code



```
#subset data
def subsetdata(data, by, subset_quantile):
    filter_standard = data.groupby([by]).size().reset_index(name='counts').counts.quantile(subset_quantile)
    subset_data = data.groupby(by).filter(lambda x: len(x) >= filter_standard)

    return filter_standard, subset_data
```

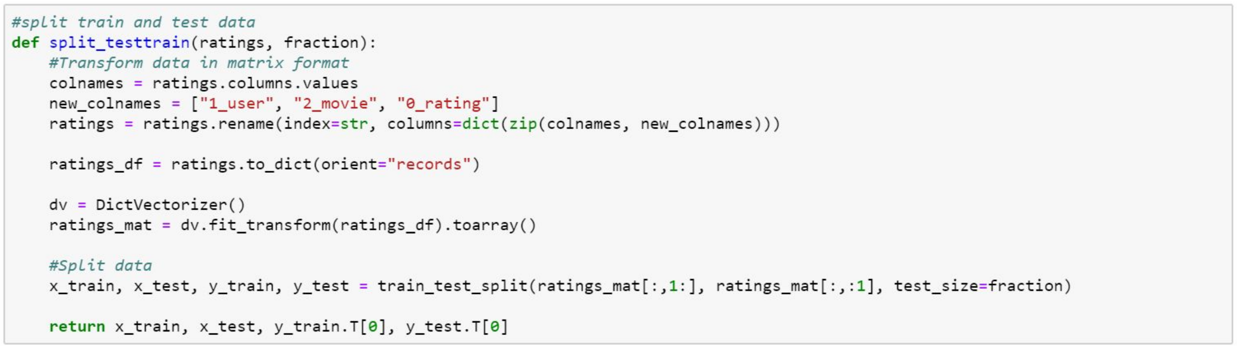
From the sample sizes, we can see that filtering prolific users and popular items returns a similar dataset in terms of sparsity. We will use these datasets to examine the performance of factorization machine models with regards to sample data sizes and sparsity.

3.2.1.2 Splitting Testing and Training data

We use the same cross validation set up as in this paper that we divide our data into a training set and a test set, where 80% of the data was used as our training set and the rest 20% as test set.

Every user-item rating pair in the original dataset was first transferred to *{"user" : ..., "item" : ..., "rating" : ...}* tuples and was stored in a list object. Then, the list of tuples was converted into our relation matrix.

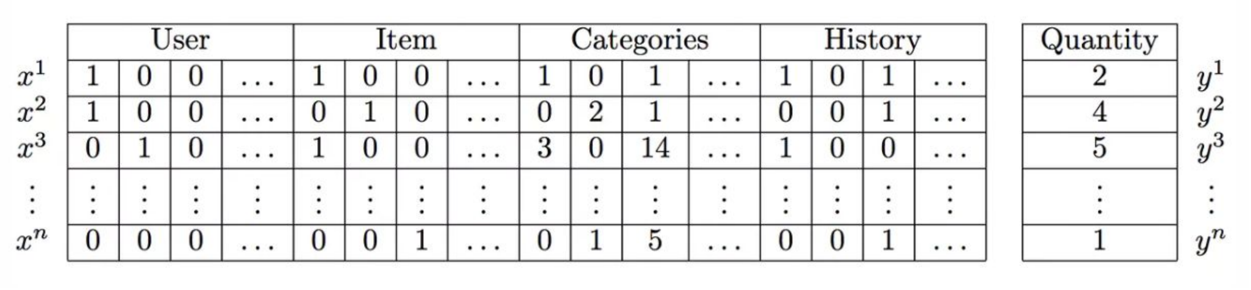
Code



```
#split train and test data
def split_testtrain(ratings, fraction):
    #Transform data in matrix format
    colnames = ratings.columns.values
    new_colnames = ["1_user", "2_movie", "0_rating"]
    ratings = ratings.rename(index=str, columns=dict(zip(colnames, new_colnames)))

    ratings_df = ratings.to_dict(orient="records")

    dv = DictVectorizer()
    ratings_mat = dv.fit_transform(ratings_df).toarray()

    #Split data
    x_train, x_test, y_train, y_test = train_test_split(ratings_mat[:,1:], ratings_mat[:,:1], test_size=fraction)

    return x_train, x_test, y_train.T[0], y_test.T[0]
```

3.2.1.3 One Hot Encoding

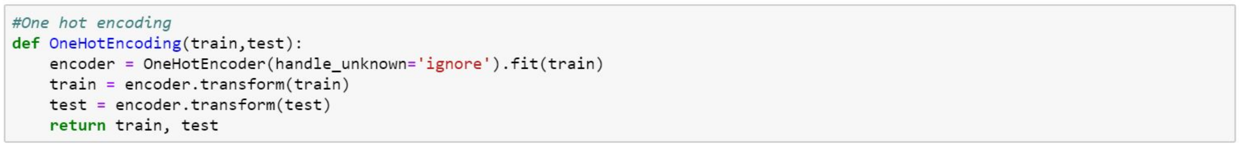
One hot encoding is a process by which categorical variables are converted into a form that could be provided to machine learning algorithms to do a better job in prediction. We followed the original implementation of Factorization Machines and structure our training data as follows:



	User				Item				Categories				History				Quantity	
x^1	1	0	0	...	1	0	0	...	1	0	1	...	1	0	1	...	2	y^1
x^2	1	0	0	...	0	1	0	...	0	2	1	...	0	0	1	...	4	y^2
x^3	0	1	0	...	1	0	0	...	3	0	14	...	1	0	0	...	5	y^3
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
x^n	0	0	0	...	0	0	1	...	0	1	5	...	0	0	1	...	1	y^n

(Source: https://getstream.io/blog/factorization-recommendation-systems/)

Code



```
#One hot encoding
def OneHotEncoding(train,test):
    encoder = OneHotEncoder(handle_unknown='ignore').fit(train)
    train = encoder.transform(train)
    test = encoder.transform(test)
    return train, test
```

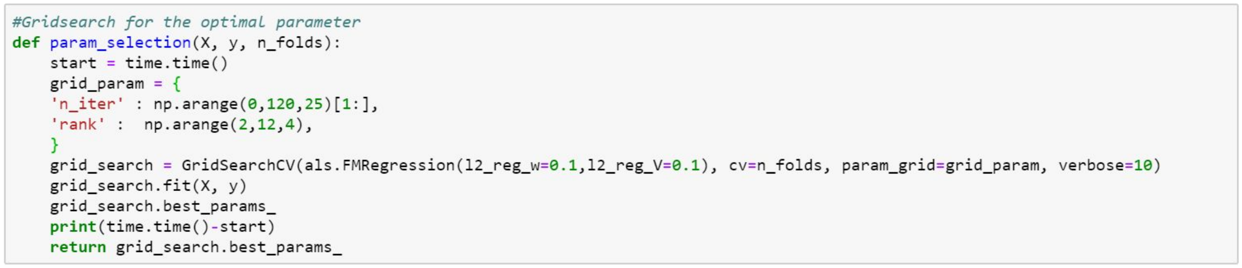
3.2.2 Hyperparameter Tuning

In machine learning, hyperparameter tuning is the problem of choosing a set of optimal hyperparameters for a learning algorithm. We will tune the parameter for each sample dataset in this model to find the optimal parameter.

The parameters we tuned here are:
1) **n_iter** – The number of samples for the MCMC sampler, iterations over the training set for ALS and number of steps for SGD.
2) **rank** – The rank of the factorization used for the second order interactions.

We used the grid search provided by GridSearchCV to exhaustively generates candidates from our grid of parameter. We also used the 3-fold cross validation as our cross-validation splitting strategy.

Code



```
#Gridsearch for the optimal parameter
def param_selection(X, y, n_folds):
    start = time.time()
    grid_param = {
    'n_iter' : np.arange(0,120,25)[1:],
    'rank' :  np.arange(2,12,4),
    }
    grid_search = GridSearchCV(als.FMRegression(l2_reg_w=0.1,l2_reg_V=0.1), cv=n_folds, param_grid=grid_param, verbose=10)
    grid_search.fit(X, y)
    grid_search.best_params_
    print(time.time()-start)
    return grid_search.best_params_
```

3.2.3 Model Fitting

We trained the factorization machine model by alternative least square (ALS) method. The algorithm we used was based on fastFM, a Python library for Factorization Machines.

The hyper parameters for each subsample are different. They are determined by the previous step of hyperparameter tuning.

3.2.4 Model Evaluation

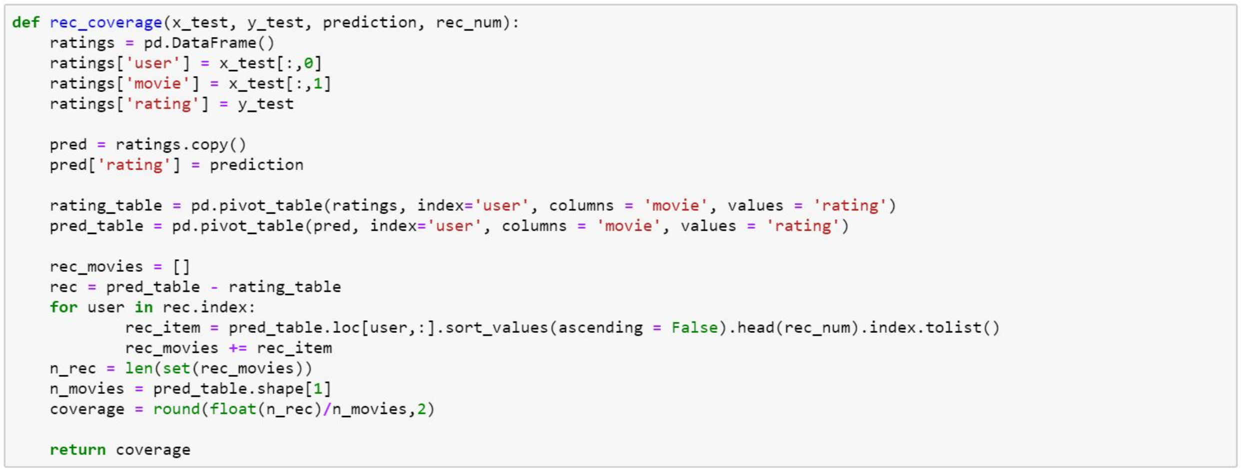
We used Root Mean Square Error (RMSE) and Mean Absolute Error (MAE) as the evaluation error metrics of the model. The two methods are the two of the most common metrics used to measure accuracy for continuous variables.

Aside from traditional accuracy metrics, we also devised an item-coverage method to test for accuracy. Item-coverage is the proportion of possible recommended items to all users. The coverage was able to offer a precise feedback about whether the recommendation covered relevant items and serendipity, rather than only contained popular items.

$$ItemCoverage = \frac{Number\ of\ items\ that\ are\ possible\ to\ be\ recommended}{Number\ of\ all\ items}$$

The calculation of RMSE and MAE are implemented with functions in package sklearn, and the calculation of item-coverage was through our self-written function.

Code



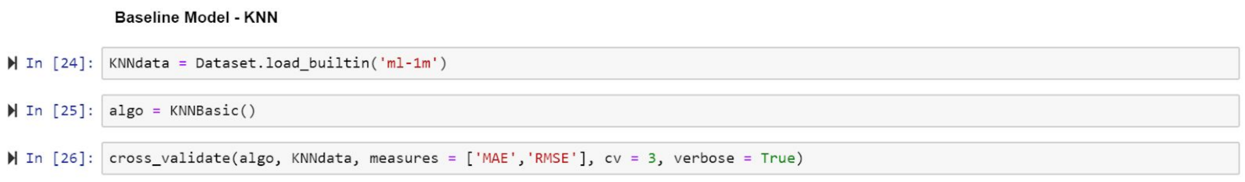
```
def rec_coverage(x_test, y_test, prediction, rec_num):
    ratings = pd.DataFrame()
    ratings['user'] = x_test[:,0]
    ratings['movie'] = x_test[:,1]
    ratings['rating'] = y_test

    pred = ratings.copy()
    pred['rating'] = prediction

    rating_table = pd.pivot_table(ratings, index='user', columns = 'movie', values = 'rating')
    pred_table = pd.pivot_table(pred, index='user', columns = 'movie', values = 'rating')

    rec_movies = []
    rec = pred_table - rating_table
    for user in rec.index:
            rec_item = pred_table.loc[user,:].sort_values(ascending = False).head(rec_num).index.tolist()
            rec_movies += rec_item
    n_rec = len(set(rec_movies))
    n_movies = pred_table.shape[1]
    coverage = round(float(n_rec)/n_movies,2)

    return coverage
```

3.2.5 Benchmark model - Collaborative Filtering Using k-Nearest Neighbors (kNN)

We used a collaborative filtering recommendation system using k-Nearest Neighbors (kNN) as our baseline model. The model is calculated using Surprise, a Python scikit library for recommender systems.

The other settings of the model are the same as our factorization machine model. We also used the 3-fold cross validation as our cross-validation splitting strategy, and we used Root Mean Square Error (RMSE) and Mean Absolute Error (MAE) as the evaluation error metrics of the model.

Code



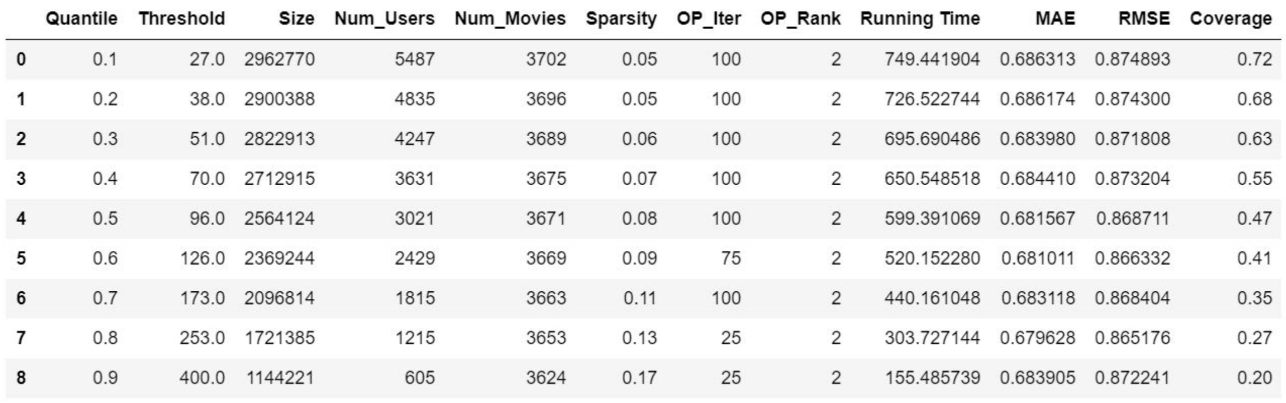
Baseline Model - KNN

```
In [24]: KNNdata = Dataset.load_builtin('ml-1m')

In [25]: algo = KNNBasic()

In [26]: cross_validate(algo, KNNdata, measures = ['MAE','RMSE'], cv = 3, verbose = True)
```

3.2.6 Evaluation

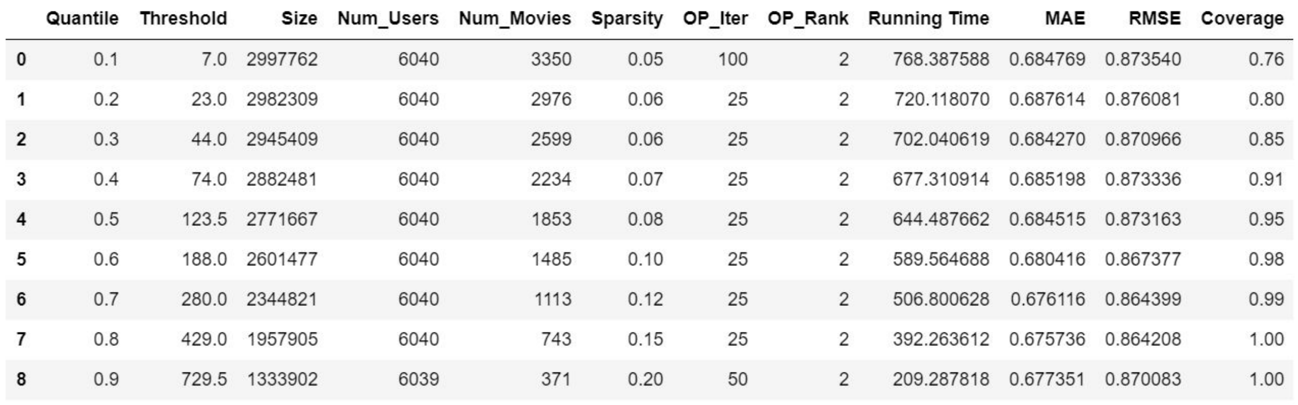
We calculated the running time, error metrics - MAE and RMSE and coverage of each subsample of each subsetting method, and the result table is as follow:

3.2.6.1 Performance - Subset data from less prolific users to prolific users



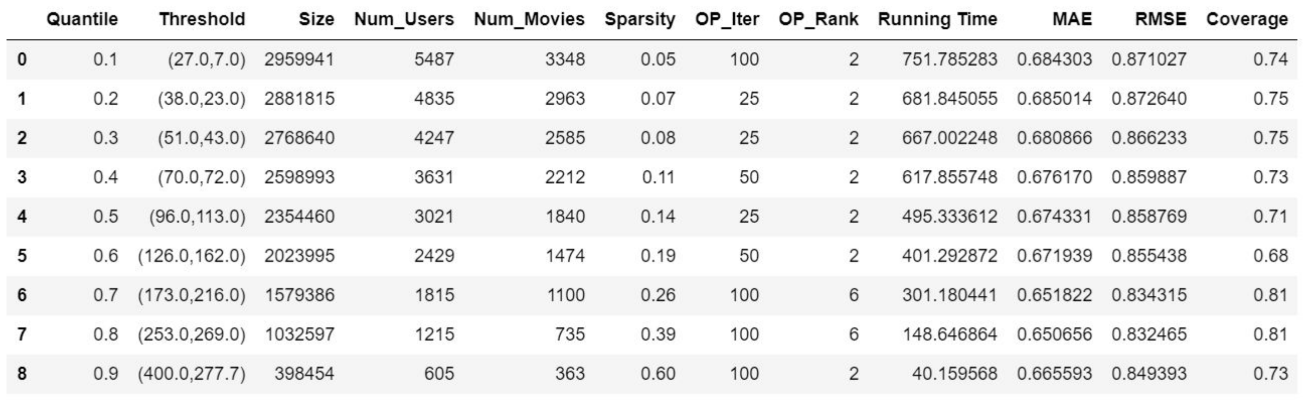
	Quantile	Threshold	Size	Num_Users	Num_Movies	Sparsity	OP_Iter	OP_Rank	Running Time	MAE	RMSE	Coverage
0	0.1	27.0	2962770	5487	3702	0.05	100	2	749.441904	0.686313	0.874893	0.72
1	0.2	38.0	2900388	4835	3696	0.05	100	2	726.522744	0.686174	0.874300	0.68
2	0.3	51.0	2822913	4247	3689	0.06	100	2	695.690486	0.683980	0.871808	0.63
3	0.4	70.0	2712915	3631	3675	0.07	100	2	650.548518	0.684410	0.873204	0.55
4	0.5	96.0	2564124	3021	3671	0.08	100	2	599.391069	0.681567	0.868711	0.47
5	0.6	126.0	2369244	2429	3669	0.09	75	2	520.152280	0.681011	0.866332	0.41
6	0.7	173.0	2096814	1815	3663	0.11	100	2	440.161048	0.683118	0.868404	0.35
7	0.8	253.0	1721385	1215	3653	0.13	25	2	303.727144	0.679628	0.865176	0.27
8	0.9	400.0	1144221	605	3624	0.17	25	2	155.485739	0.683905	0.872241	0.20

3.2.6.2 Performance - Subset data from less popular items to popular items



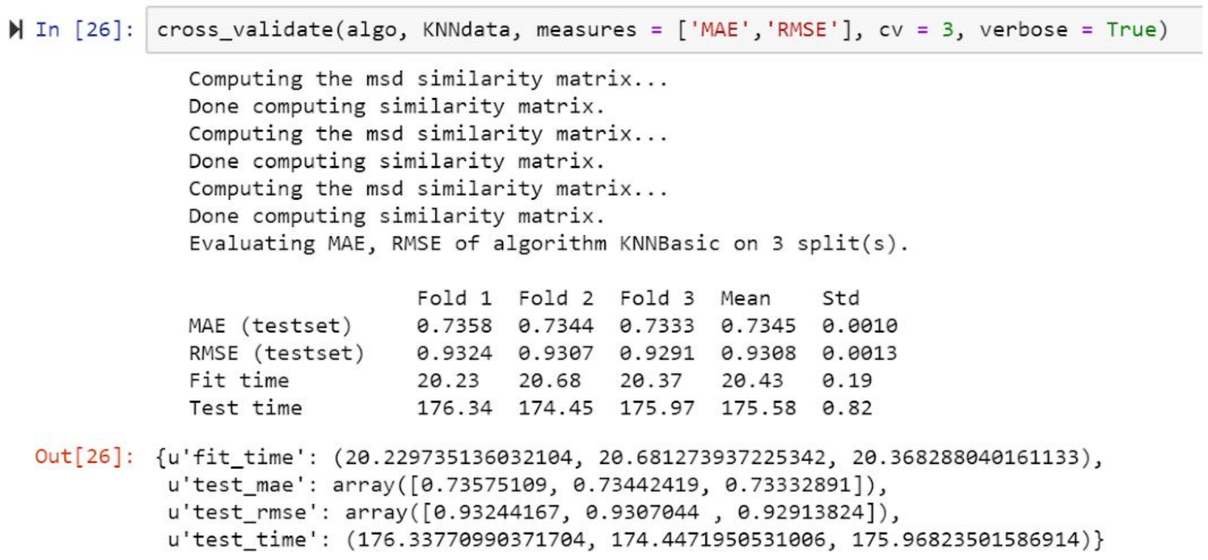
	Quantile	Threshold	Size	Num_Users	Num_Movies	Sparsity	OP_Iter	OP_Rank	Running Time	MAE	RMSE	Coverage
0	0.1	7.0	2997762	6040	3350	0.05	100	2	768.387588	0.684769	0.873540	0.76
1	0.2	23.0	2982309	6040	2976	0.06	25	2	720.118070	0.687614	0.876081	0.80
2	0.3	44.0	2945409	6040	2599	0.06	25	2	702.040619	0.684270	0.870966	0.85
3	0.4	74.0	2882481	6040	2234	0.07	25	2	677.310914	0.685198	0.873336	0.91
4	0.5	123.5	2771667	6040	1853	0.08	25	2	644.487662	0.684515	0.873163	0.95
5	0.6	188.0	2601477	6040	1485	0.10	25	2	589.564688	0.680416	0.867377	0.98
6	0.7	280.0	2344821	6040	1113	0.12	25	2	506.800628	0.676116	0.864399	0.99
7	0.8	429.0	1957905	6040	743	0.15	25	2	392.263612	0.675736	0.864208	1.00
8	0.9	729.5	1333902	6039	371	0.20	50	2	209.287818	0.677351	0.870083	1.00

3.2.6.3 Performance - Subset data in both user and item directions



	Quantile	Threshold	Size	Num_Users	Num_Movies	Sparsity	OP_Iter	OP_Rank	Running Time	MAE	RMSE	Coverage
0	0.1	(27.0,7.0)	2959941	5487	3348	0.05	100	2	751.785283	0.684303	0.871027	0.74
1	0.2	(38.0,23.0)	2881815	4835	2963	0.07	25	2	681.845055	0.685014	0.872640	0.75
2	0.3	(51.0,43.0)	2768640	4247	2585	0.08	25	2	667.002248	0.680866	0.866233	0.75
3	0.4	(70.0,72.0)	2598993	3631	2212	0.11	50	2	617.855748	0.676170	0.859887	0.73
4	0.5	(96.0,113.0)	2354460	3021	1840	0.14	25	2	495.333612	0.674331	0.858769	0.71
5	0.6	(126.0,162.0)	2023995	2429	1474	0.19	50	2	401.292872	0.671939	0.855438	0.68
6	0.7	(173.0,216.0)	1579386	1815	1100	0.26	100	6	301.180441	0.651822	0.834315	0.81
7	0.8	(253.0,269.0)	1032597	1215	735	0.39	100	6	148.646864	0.650656	0.832465	0.81
8	0.9	(400.0,277.7)	398454	605	363	0.60	100	2	40.159568	0.665593	0.849393	0.73

3.2.6.4 Performance - Benchmark model: Collaborative Filtering Using kNN

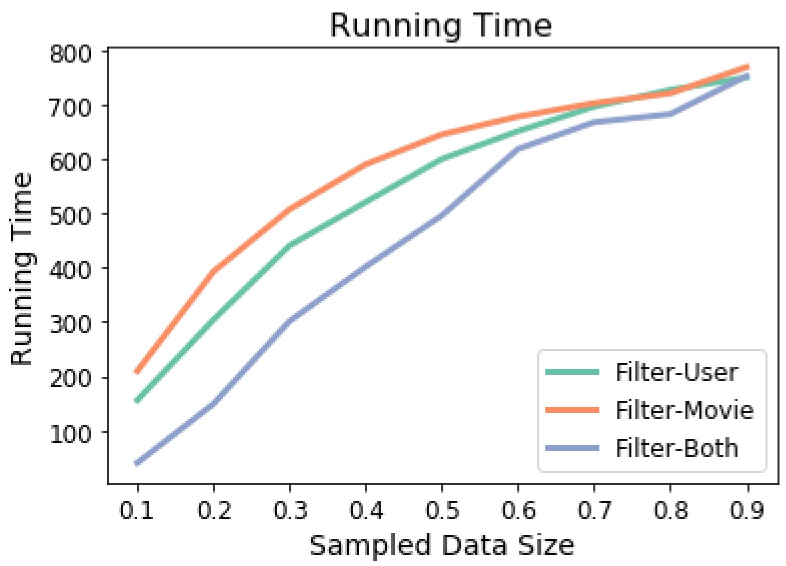


```
In [26]: cross_validate(algo, KNNdata, measures = ['MAE','RMSE'], cv = 3, verbose = True)

    Computing the msd similarity matrix...
    Done computing similarity matrix.
    Computing the msd similarity matrix...
    Done computing similarity matrix.
    Computing the msd similarity matrix...
    Done computing similarity matrix.
    Evaluating MAE, RMSE of algorithm KNNBasic on 3 split(s).

                      Fold 1  Fold 2  Fold 3  Mean    Std
    MAE (testset)     0.7358  0.7344  0.7333  0.7345  0.0010
    RMSE (testset)    0.9324  0.9307  0.9291  0.9308  0.0013
    Fit time          20.23   20.68   20.37   20.43   0.19
    Test time         176.34  174.45  175.97  175.58  0.82

Out[26]: {u'fit_time': (20.229735136032104, 20.681273937225342, 20.368288040161133),
          u'test_mae': array([0.73575109, 0.73442419, 0.73332891]),
          u'test_rmse': array([0.93244167, 0.9307044 , 0.92913824]),
          u'test_time': (176.33770990371704, 174.4471950531006, 175.96823501586914)}
```

3.2.6.5 Evaluation - Running Time

We sampled our dataset by percentile and tested the running time, and the result is as the following graph.

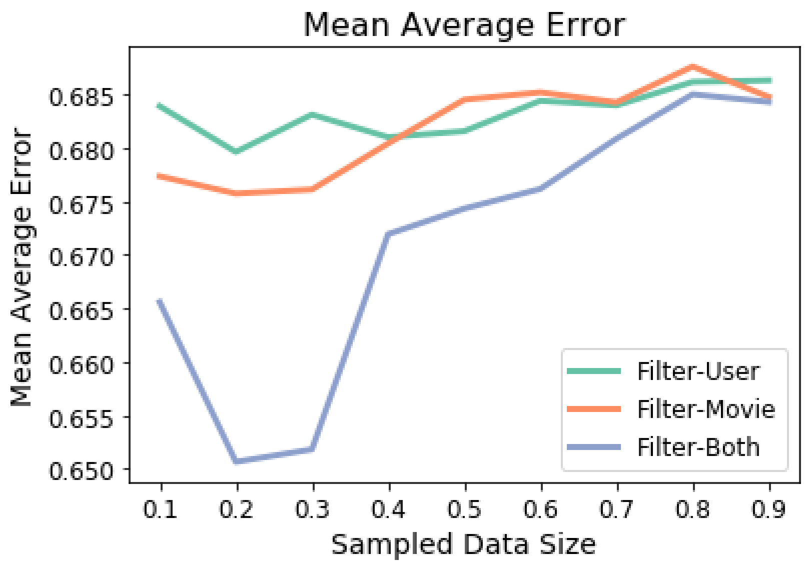
We can see that all three method's running time scale pretty well as the data size increases, following a linear trend without drastic leap. The three methods' running time also converges when the sample set gets larger. The running time is also a lot shorter than the collaborative filtering algorithm we implemented earlier in homework 2, that even running the entire dataset takes around 800 seconds.



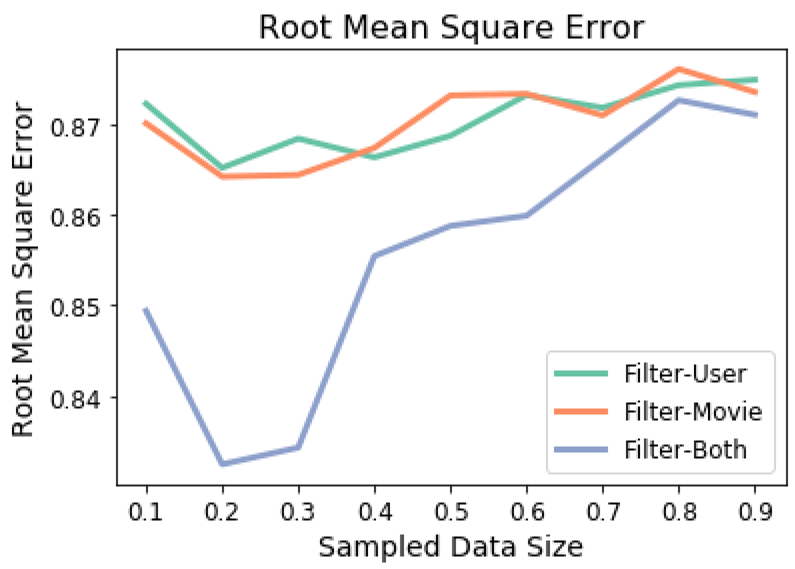
3.2.6.6 Evaluation - Accuracy: Mean Average Error(MAE), Root Mean Square Error (RMSE)

We sampled our dataset by percentile and tested the accuracy by MAE and RMSE metric, and the result is as the following graph.

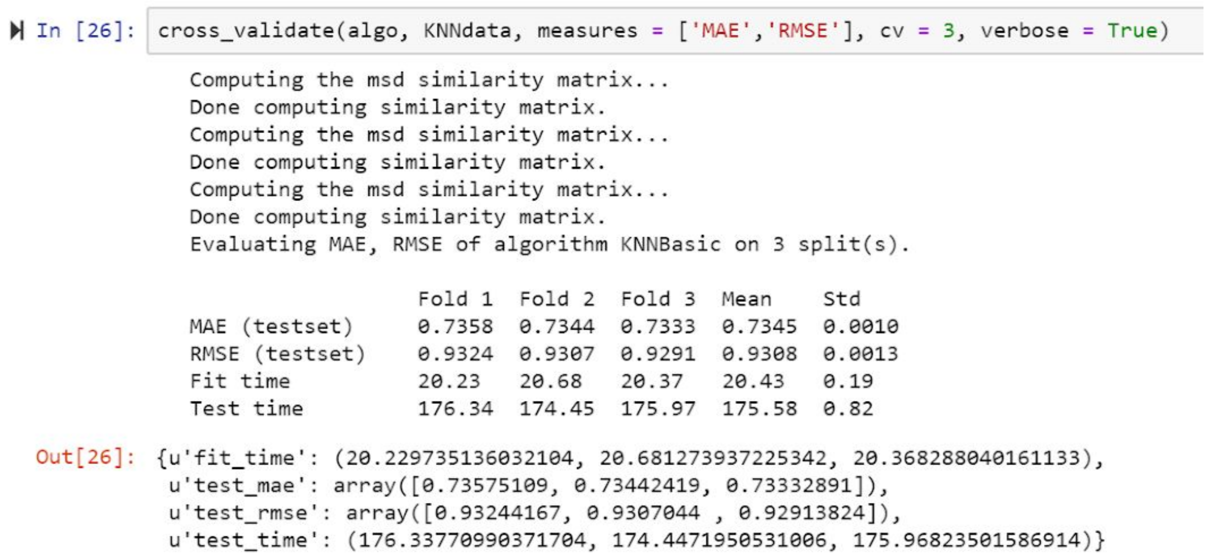
We can see that the MAE of filtering both the user and movie is significantly lower than filtering only with user or movie. While all three error metric show the trend to first decrease then increase, the models that are filtered only by user or movie have a relatively stable MAE regardless of the sample data size and sparsity. The MAE of filtering with both user and item has a more turbulent movement in MAE a the subsample data size grow larger and sparser.



The same trend can also be found in the RMSE graph, that the models filtered by user or item only don't vary much in error and are quite stable regardless of data size and sparsity.



We then compare the error metric result with our benchmark model - Collaborative Filtering Using k-Nearest Neighbors (kNN) .

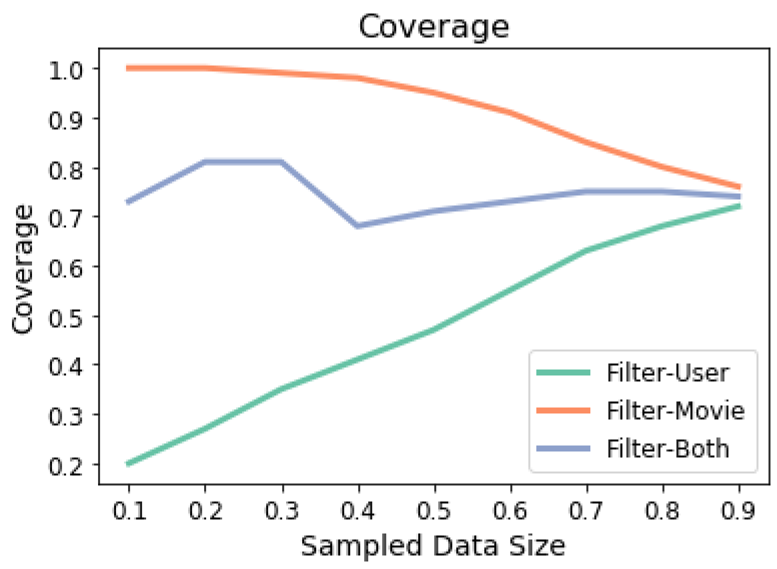


```
In [26]: cross_validate(algo, KNNdata, measures = ['MAE','RMSE'], cv = 3, verbose = True)

    Computing the msd similarity matrix...
    Done computing similarity matrix.
    Computing the msd similarity matrix...
    Done computing similarity matrix.
    Computing the msd similarity matrix...
    Done computing similarity matrix.
    Evaluating MAE, RMSE of algorithm KNNBasic on 3 split(s).

                      Fold 1  Fold 2  Fold 3  Mean    Std
    MAE (testset)     0.7358  0.7344  0.7333  0.7345  0.0010
    RMSE (testset)    0.9324  0.9307  0.9291  0.9308  0.0013
    Fit time          20.23   20.68   20.37   20.43   0.19
    Test time         176.34  174.45  175.97  175.58  0.82

Out[26]: {u'fit_time': (20.229735136032104, 20.681273937225342, 20.368288040161133),
          u'test_mae': array([0.73575109, 0.73442419, 0.73332891]),
          u'test_rmse': array([0.93244167, 0.9307044 , 0.92913824]),
          u'test_time': (176.33770990371704, 174.4471950531006, 175.96823501586914)}
```

The lowest MAE we got from KNN-CF model is 0.73, and the lowest RMSE is 0.92. We can see from our graph that all of our subsamples in all subsetting methods have a lower MAE and RMSE than the benchmark model, showing that our factorization machine model is more accurate then our benchmark KNN-CF model.

3.2.6.7 Evaluation - Coverage

We sampled our dataset by percentile and tested the accuracy by item-coverage, and the result is as the following graph.



We see that the three subsetting methods show very different trend in coverage. The first subsetting method by user works better with more sparse data, and the second subsetting method by movie works better with denser data. The third subsetting method that filters both the user and movie has the most stable coverage, and the coverage is relatively high regardless of data size and sparsity.

3.2.7 Results and Conclusion

Our first model is a factorization machine model without side information. We subsampled datasets from small to large, and subsampled users/items from sparsely-populated to well-populated. We used three subsetting methods to fulfill this, that we tried subsetting the data by how prolific the user is, how popular the movie is, and one taking both into account.

We find that if we subset the data with just taking user or item into account, the factorization machine model will not perform well. Both the MAE and RMSE are high in this case regardless of data size or sparsity. However, its accuracy is still higher than the traditional KNN-CF model that predicts users’ ratings through similarities.

If we subset the data by both user and item’s feature, the model works equally well for both users and items, and it works slightly better with denser data. The accuracy is also higher than the traditional KNN-CF model that predicts users’ ratings through similarities.

Using item-coverage to test for quality gives us a new and interesting result. We find that as less popular items increase in our sample, the item-coverage decreases. However, if we have more less prolific users in our sample, the item-coverage increases. With this information in mind, we should subset our data more carefully considering the features of our data. If we are working with more sparse datasets, we might want to consider subsetting data with just user, and if we are working with denser datasets, we might want to consider subsetting data with just item. We also might consider to filter both the user and item's activeness, as the coverage of this method is more stable and remains relatively high regardless of data size and sparsity.

3.3 Second model - Factorization Machine with side information

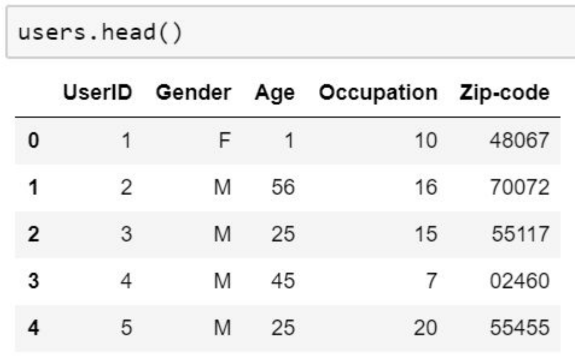
After analyzing factorization machine model with no side information and only user and item ratings, our second model incorporates side information about users and items into factorization machine model to see if the accuracy will be improved.

3.3.1 Data Preprocessing

We preprocessed user's and item's information and incorporated them into the model. The other way of subsetting data, splitting test and train data, and one hot encoding data are the same as in the first model.

3.3.1.1 User's information

User information is in the file "users.dat" from MovieLens data and is in the following format: UserID::Gender::Age::Occupation::Zip-code.



```
users.head()
```

	UserID	Gender	Age	Occupation	Zip-code
0	1	F	1	10	48067
1	2	M	56	16	70072
2	3	M	25	15	55117
3	4	M	45	7	02460
4	5	M	25	20	55455

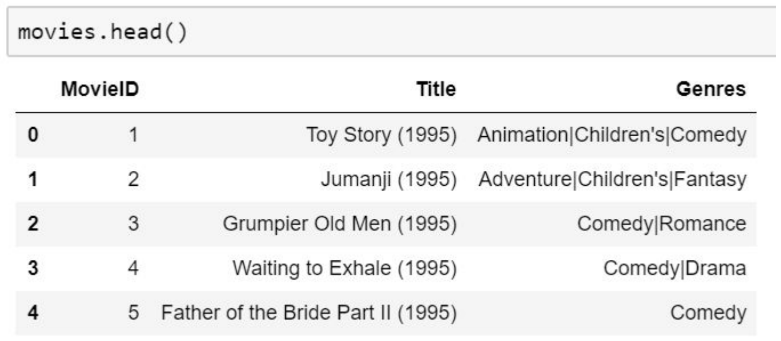
The format follows:
- **Gender** is denoted by a "M" for male and "F" for female
- **Age** is chosen from the following ranges:
* 1: "Under 18" * 18: "18-24" * 25: "25-34" * 35: "35-44" * 45: "45-49 * 50: "50-55" * 56: "56+"
- **Occupation** is chosen from the following choices:
* 0: "other" or not specified * 1: "academic/educator" * 2: "artist" * 3: "clerical/admin

* 4: "college/grad student" * 5: "customer service" * 6: "doctor/health care"
* 7: "executive/managerial" * 8: "farmer" * 9: "homemaker" * 10: "K-12 student"
* 11: "lawyer" * 12: "programmer" * 13: "retired" * 14: "sales/marketing"
* 15: "scientist" * 16: "self-employed" * 17: "technician/engineer"
* 18: "tradesman/craftsman" * 19: "unemployed" * 20: "writer"

To use one hot encoding, we need to recode every categorical data into numeric representation. Since the field zip-code comes in too many different formats, we decided to drop the variable off. We then recode Gender into 0 and 1, where 0 represents female and 1 represents male.

3.3.1.2 Movie's information

Movie information is in the file "movies.dat" from MovieLens data and is in the following format: MovieID::Title::Genres



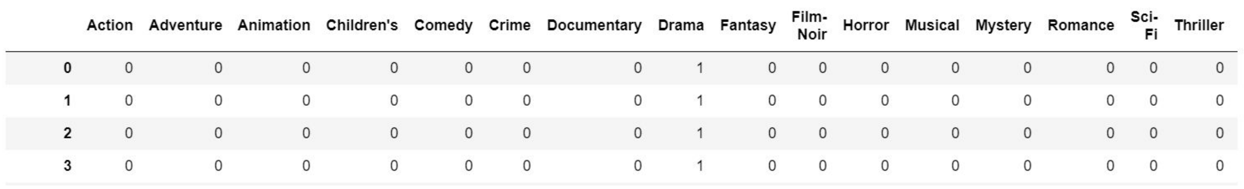
```
movies.head()
```

	MovieID	Title	Genres
0	1	Toy Story (1995)	Animation|Children's|Comedy
1	2	Jumanji (1995)	Adventure|Children's|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama
4	5	Father of the Bride Part II (1995)	Comedy

The format follows:
- **Titles** are identical to titles provided by the IMDB (including year of release)
- **Genres** are pipe-separated and are selected from the following genres:
* Action * Adventure * Animation * Children's * Comedy * Crime * Documentary
* Drama * Fantasy * Film-Noir *Horror * Musical * Mystery * Romance * Sci-Fi
* Thriller * War * Western

We can see that the "Genres" category is not processable by the factorization machine, therefore, we need to preprocess it by removing the punctuation and expanding the genres into dataset.

We will represent each genre as a dummy variable. If the row entry's movie belongs to that category, the entry would be 1, otherwise 0.



	Action	Adventure	Animation	Children's	Comedy	Crime	Documentary	Drama	Fantasy	Film-Noir	Horror	Musical	Mystery	Romance	Sci-Fi	Thriller
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

3.3.2 Training process

The training of factorization machine in this model is the same as in the first model. However, we will not tune the parameters but use the result we obtained from the previous model directly. The reason is that we discovered the hyperparameter for each subsample within the same subset method doesn’t vary much. Therefore, we can save the running time of tuning them and use it directly to train our factorization machine model by alternative least square (ALS) method.

We will also use Root Mean Square Error (RMSE), Mean Absolute Error (MAE) and item-coverage as the evaluation error metrics of the model.

We will also use a collaborative filtering recommendation system using k-Nearest Neighbors (kNN) as our baseline model.The validation splitting strategy is also set to 3-fold.

3.3.3 Evaluation

We calculated the running time, error metrics - MAE and RMSE and coverage of each subsample of each subsetting method, and the result table is as follow:

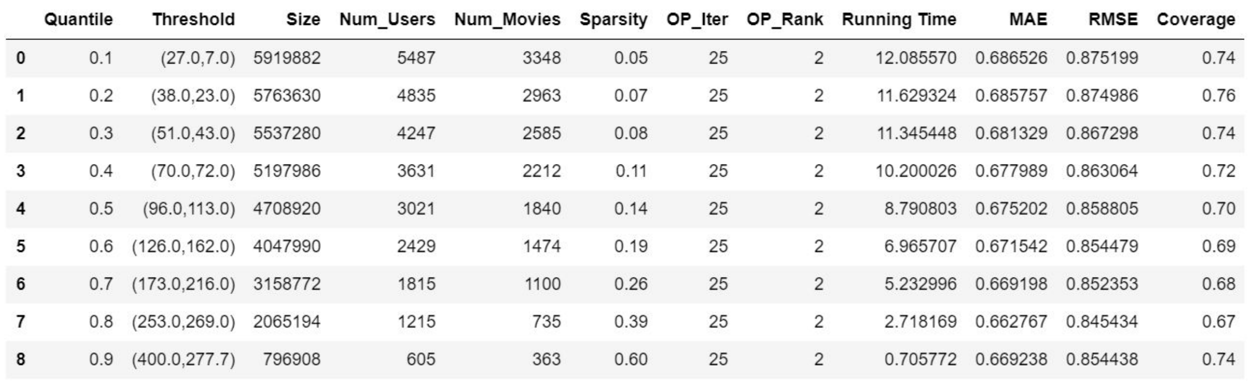
3.3.3.1 Performance - Subset data from less prolific users to prolific users

	Quantile	Threshold	Size	Num_Users	Num_Movies	Sparsity	OP_Iter	OP_Rank	Running Time	MAE	RMSE	Coverage
0	0.1	27.0	23702160	5487	3702	0.05	100	2	139.628221	0.688364	0.877603	1.0
1	0.2	38.0	23203104	4835	3696	0.05	100	2	136.748012	0.684637	0.872970	1.0
2	0.3	51.0	22583304	4247	3689	0.06	100	2	133.654470	0.684583	0.871929	1.0
3	0.4	70.0	21703320	3631	3675	0.07	100	2	123.384483	0.683063	0.870505	1.0
4	0.5	96.0	20512992	3021	3671	0.08	100	2	118.129110	0.681535	0.867517	1.0
5	0.6	126.0	18953952	2429	3669	0.09	100	2	104.969195	0.678930	0.864937	1.0
6	0.7	173.0	16774512	1815	3663	0.11	100	2	91.671748	0.678836	0.864744	1.0
7	0.8	253.0	13771080	1215	3653	0.13	100	2	72.089246	0.677348	0.863175	1.0
8	0.9	400.0	9153768	605	3624	0.17	100	2	43.558324	0.682891	0.871443	1.0

3.3.3.2 Performance - Subset data from less popular items to popular items

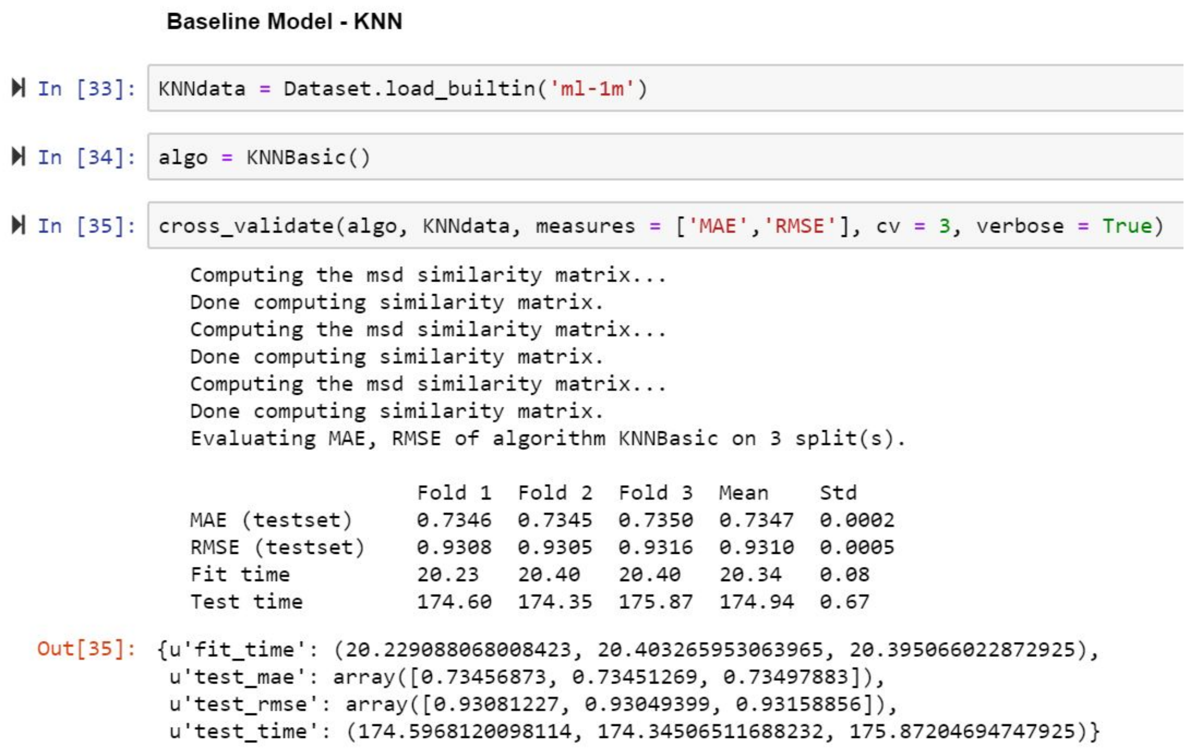
	Quantile	Threshold	Size	Num_Users	Num_Movies	Sparsity	OP_Iter	OP_Rank	Running Time	MAE	RMSE	Coverage
0	0.1	7.0	5995524	6040	3350	0.05	25	2	12.236929	0.689428	0.879453	0.76
1	0.2	23.0	5964618	6040	2976	0.06	25	2	12.064855	0.687384	0.875639	0.81
2	0.3	44.0	5890818	6040	2599	0.06	25	2	11.978348	0.682721	0.869367	0.86
3	0.4	74.0	5764962	6040	2234	0.07	25	2	11.581689	0.688576	0.877720	0.91
4	0.5	123.5	5543334	6040	1853	0.08	25	2	10.961639	0.680661	0.867589	0.95
5	0.6	188.0	5202954	6040	1485	0.10	25	2	10.100358	0.680511	0.867270	0.98
6	0.7	280.0	4689642	6040	1113	0.12	25	2	8.670737	0.678078	0.864230	0.99
7	0.8	429.0	3915810	6040	743	0.15	25	2	6.848009	0.675242	0.862520	1.00
8	0.9	729.5	2667804	6039	371	0.20	25	2	4.067644	0.671836	0.864080	1.00

3.3.3.3 Performance - Subset data in both user and item directions



	Quantile	Threshold	Size	Num_Users	Num_Movies	Sparsity	OP_Iter	OP_Rank	Running Time	MAE	RMSE	Coverage
0	0.1	(27.0,7.0)	5919882	5487	3348	0.05	25	2	12.085570	0.686526	0.875199	0.74
1	0.2	(38.0,23.0)	5763630	4835	2963	0.07	25	2	11.629324	0.685757	0.874986	0.76
2	0.3	(51.0,43.0)	5537280	4247	2585	0.08	25	2	11.345448	0.681329	0.867298	0.74
3	0.4	(70.0,72.0)	5197986	3631	2212	0.11	25	2	10.200026	0.677989	0.863064	0.72
4	0.5	(96.0,113.0)	4708920	3021	1840	0.14	25	2	8.790803	0.675202	0.858805	0.70
5	0.6	(126.0,162.0)	4047990	2429	1474	0.19	25	2	6.965707	0.671542	0.854479	0.69
6	0.7	(173.0,216.0)	3158772	1815	1100	0.26	25	2	5.232996	0.669198	0.852353	0.68
7	0.8	(253.0,269.0)	2065194	1215	735	0.39	25	2	2.718169	0.662767	0.845434	0.67
8	0.9	(400.0,277.7)	796908	605	363	0.60	25	2	0.705772	0.669238	0.854438	0.74

3.3.3.4 Performance - Benchmark model: Collaborative Filtering Using kNN



Baseline Model - KNN

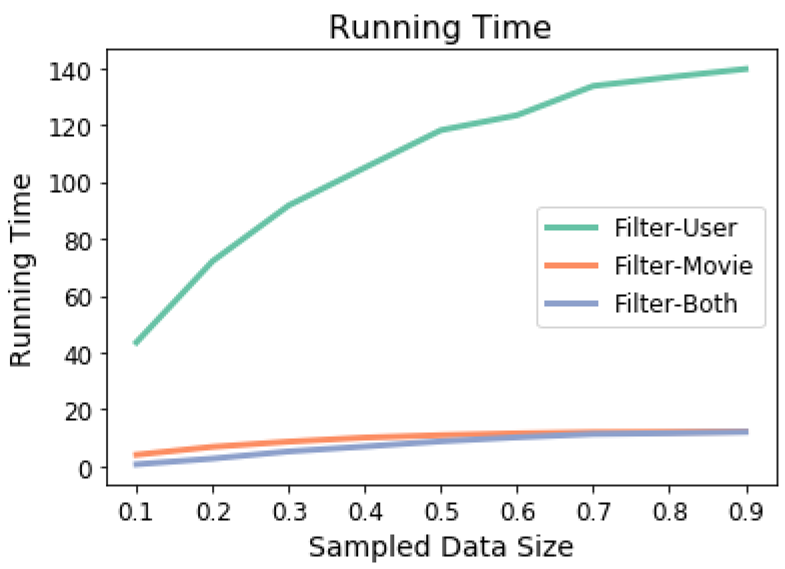
```
In [33]: KNNdata = Dataset.load_builtin('ml-1m')

In [34]: algo = KNNBasic()

In [35]: cross_validate(algo, KNNdata, measures = ['MAE','RMSE'], cv = 3, verbose = True)

    Computing the msd similarity matrix...
    Done computing similarity matrix.
    Computing the msd similarity matrix...
    Done computing similarity matrix.
    Computing the msd similarity matrix...
    Done computing similarity matrix.
    Evaluating MAE, RMSE of algorithm KNNBasic on 3 split(s).

                      Fold 1  Fold 2  Fold 3  Mean    Std
    MAE (testset)     0.7346  0.7345  0.7350  0.7347  0.0002
    RMSE (testset)    0.9308  0.9305  0.9316  0.9310  0.0005
    Fit time          20.23   20.40   20.40   20.34   0.08
    Test time         174.60  174.35  175.87  174.94  0.67

Out[35]: {u'fit_time': (20.229088068008423, 20.403265953063965, 20.395066022872925),
          u'test_mae': array([0.73456873, 0.73451269, 0.73497883]),
          u'test_rmse': array([0.93081227, 0.93049399, 0.93158856]),
          u'test_time': (174.5968120098114, 174.34506511688232, 175.87204694747925)}
```

3.3.3.5 Evaluation - Running Time

We sampled our dataset by percentile and tested the running time, and the result is as the following graph.

We can see that all three method's running time scale pretty well as the data size increases, following a linear trend without drastic leap. The running time of filtering by user is a lot longer since it takes 100 iterations to reach the optimal error.

Compared to the first model, we see that the running time of filtering by movie and filtering by both user and movie doesn't vary much after we incorporated side information.

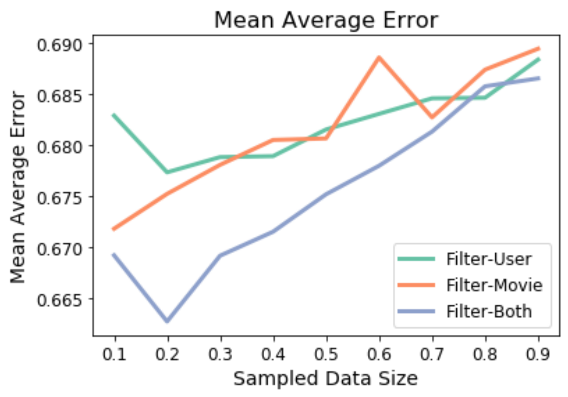


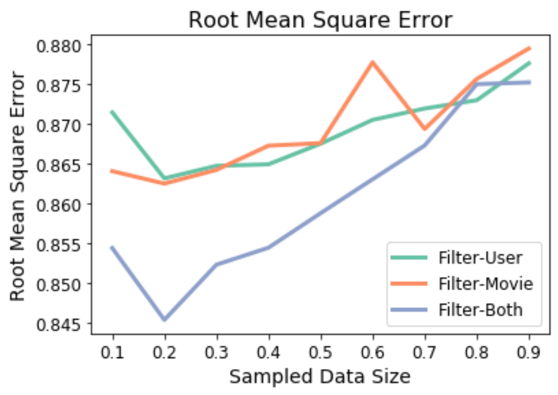
3.3.3.6 Evaluation - Accuracy: Mean Average Error(MAE), Root Mean Square Error (RMSE)

We sampled our dataset by percentile and tested the accuracy by MAE and RMSE metric, and the result is as the following graph.

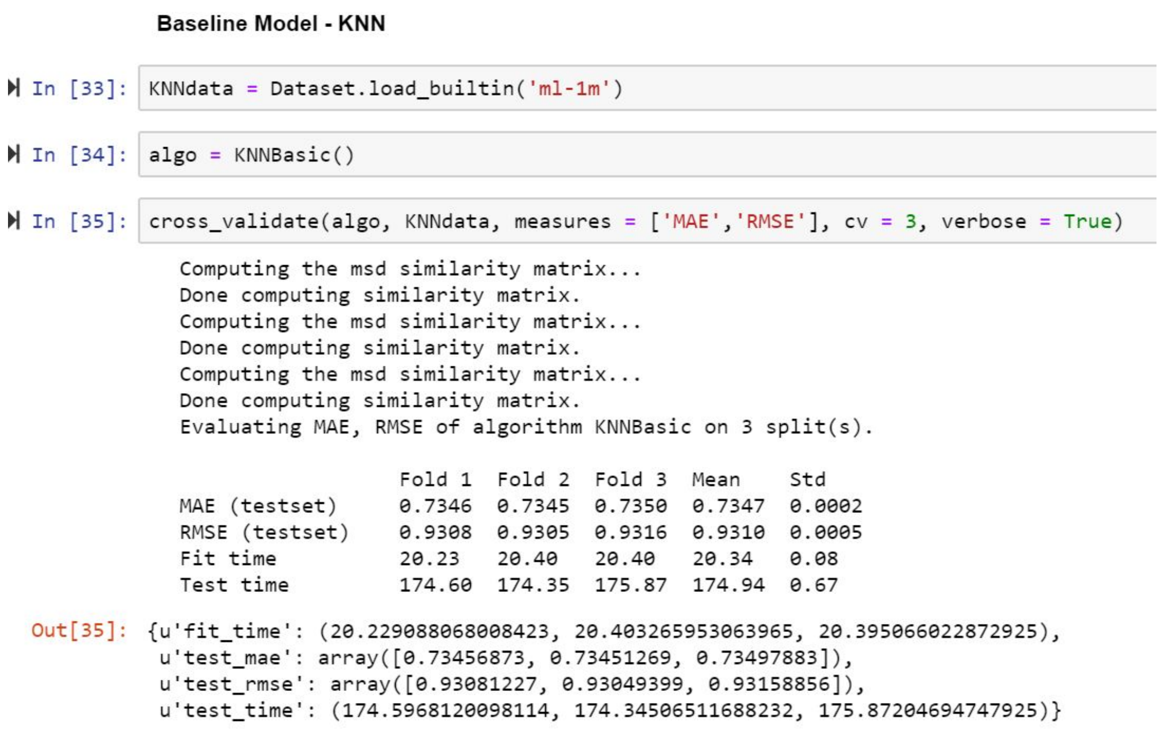
We can see that like the previous model, the MAE of filtering both the user and movie is significantly lower than filtering only with user or movie. MAE of filtering by user is slightly higher, but a lot more stable compared to the other methods.

The same trend can also be found in the RMSE graph.





We then compare the error metric result with our benchmark model - Collaborative Filtering Using k-Nearest Neighbors (kNN) .



Baseline Model - KNN

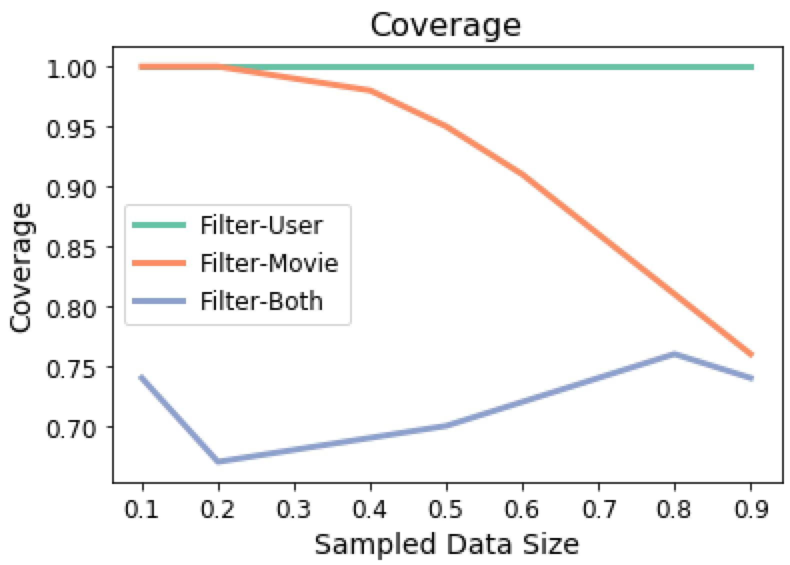
```
In [33]: KNNdata = Dataset.load_builtin('ml-1m')

In [34]: algo = KNNBasic()

In [35]: cross_validate(algo, KNNdata, measures = ['MAE','RMSE'], cv = 3, verbose = True)

    Computing the msd similarity matrix...
    Done computing similarity matrix.
    Computing the msd similarity matrix...
    Done computing similarity matrix.
    Computing the msd similarity matrix...
    Done computing similarity matrix.
    Evaluating MAE, RMSE of algorithm KNNBasic on 3 split(s).

                      Fold 1  Fold 2  Fold 3  Mean    Std
    MAE (testset)     0.7346  0.7345  0.7350  0.7347  0.0002
    RMSE (testset)    0.9308  0.9305  0.9316  0.9310  0.0005
    Fit time          20.23   20.40   20.40   20.34   0.08
    Test time         174.60  174.35  175.87  174.94  0.67

Out[35]: {u'fit_time': (20.229088068008423, 20.403265953063965, 20.395066022872925),
          u'test_mae': array([0.73456873, 0.73451269, 0.73497883]),
          u'test_rmse': array([0.93081227, 0.93049399, 0.93158856]),
          u'test_time': (174.5968120098114, 174.34506511688232, 175.87204694747925)}
```

The lowest MAE we got from KNN-CF model is 0.73, and the lowest RMSE is 0.93. We can see from our graph that all of our subsamples in all subsetting methods have a lower MAE and RMSE than the benchmark model, showing that our factorization machine model is more accurate then our benchmark KNN-CF model.

3.3.3.7 Evaluation - Coverage

We sampled our dataset by percentile and tested the accuracy by item-coverage, and the result is as the following graph.

We see that the three subsetting methods show very different trend in coverage. The first subsetting method by user is very stable across from all data sizes and sparsity. It also returns a really high item-coverage. The second subsetting method by movie works better with denser data, which matches with our findings in the previous model. The third subsetting method that filters both the user and movie has the lowest coverage, and doesn't seem to improve/affect much by data size and sparsity. It is quite different from what we see in the previous model without side information.



3.3.4 Results and Conclusion

Our second model is a factorization machine model with side information. We subsampled datasets from small to large, and subsampled users/items from sparsely-populated to well-populated. We used three subsetting methods to fulfill this, that we tried subsetting the data by how prolific the user is, how popular the movie is, and one taking both into account.

We find that if we subset the data by item, the factorization machine model will not perform well. The error would be higher and very turbulent compared to other subsetting methods. However, its accuracy is still higher than the traditional KNN-CF model that predicts users' ratings through similarities.

If we subset the data by both user and item's feature, the model works equally well for both users and items, and it works slightly better with denser data. The accuracy is also higher than the traditional KNN-CF model that predicts users' ratings through similarities.

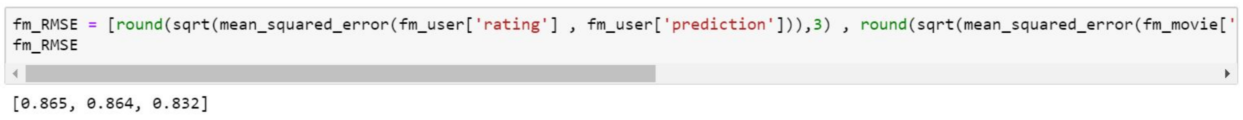
Using item-coverage to test for quality gives us a new and interesting result. We find that as less popular items increase in our sample, the item-coverage decreases. With this information in mind, if we are working with denser datasets, we might want to consider subsetting data with just item. If we filter both the user and item's activeness, the coverage of this method is significantly lower and doesn't seem to improve regardless of data size and sparsity. The best subsetting method seems to be filtering by user's activeness, that whether the users are prolific or not doesn't affect the item-coverage as long as we have the side information of the users and the items.

3.4 Model Extension

To try to improve the model, we tried to build a hybrid model combining the two factorization models we explored.

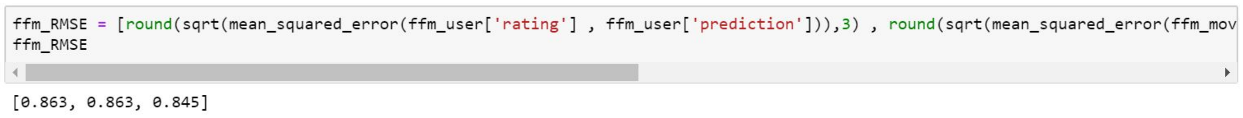
We combined the ratings by averaging predicted values from the two models. We then weighted the proportion of the model based on the RMSE result. We computed the predicted ratings by a weighted average result of the two factorization machine models, where the weight is inverse proportional to the individual RMSE of each model.

However, we didn't successfully improved the accuracy by fusing the two models together. We think that it might be the reason that the two models are too similar, so that it wouldn't have much effect when we merge the two models together. It can also be possible that there are better ways of merging and constructing hybrid models other that weighting it by its RMSE value respectfully.



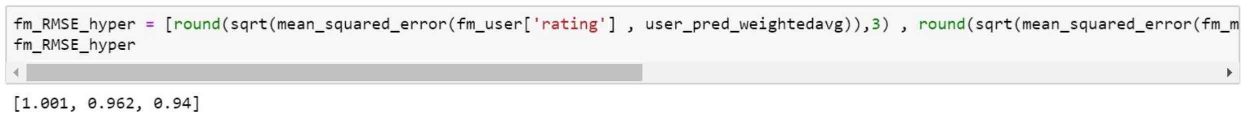
```
fm_RMSE = [round(sqrt(mean_squared_error(fm_user['rating'] , fm_user['prediction'])),3) , round(sqrt(mean_squared_error(fm_movie['
fm_RMSE
```

```
[0.865, 0.864, 0.832]
```



```
ffm_RMSE = [round(sqrt(mean_squared_error(ffm_user['rating'] , ffm_user['prediction'])),3) , round(sqrt(mean_squared_error(ffm_mov
ffm_RMSE
```

```
[0.863, 0.863, 0.845]
```



```
fm_RMSE_hyper = [round(sqrt(mean_squared_error(fm_user['rating'] , user_pred_weightedavg)),3) , round(sqrt(mean_squared_error(fm_m
fm_RMSE_hyper
```

```
[1.001, 0.962, 0.94]
```

4 - Wide and Deep Learning Recommendation System

Wide and Deep learning are a set of Recommendation systems which have been shown to perform much better. We have seen that memorization of feature interactions is possible through the use of the cross product of feature interactions while generalization requires more feature engineering effort. Deep neural networks are able to generalize using low dimensional embeddings even with very little feature engineering. But, thit tends to cause the neural network to overgeneralize and recommend lower relevance items. Thus the wide and deep learning tries to leverage both the methods and tries to leverage both memorization and generalization.

4.1 Objective

We built two types of wide and deep recommendation systems, one with and one without user and item features of side information. Our objective is to compare the two models to evaluate whether incorporating side information would improve the accuracy of Wide and deep models. we are also interested in knowing the model's performance in running time and accuracy when we subsample datasets from small to large and users/items from sparsely-populated to well-populated.

The accuracy metrics we used in evaluating the recommendation systems are Root Mean Square Error (RMSE) and Mean Absolute Error (MAE) . The two methods are the two of the most common metrics used to measure accuracy for continuous variables.

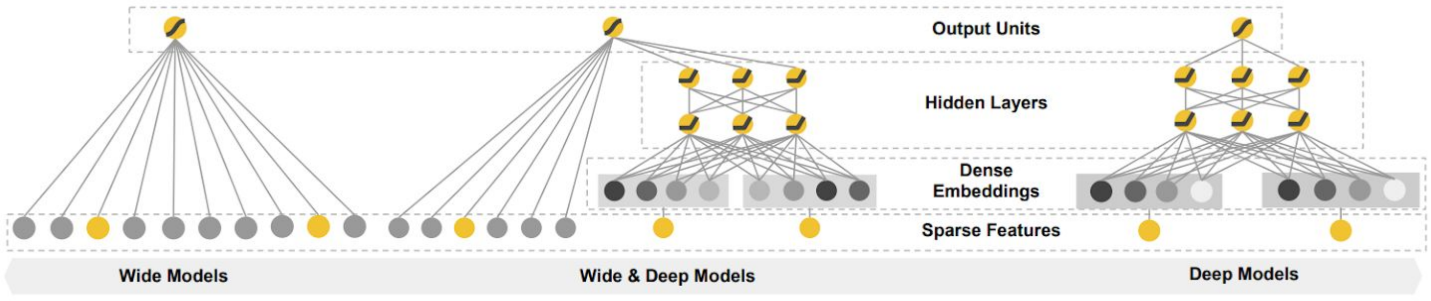


Figure: Spectrum of wide and deep models

4.2 Wide and deep learning with side information

In this model we use the wide and deep network with the side information available about the users and the movies.

4.2.1 Data Preprocessing

The data preprocessing consists of three parts - splitting test and train data, converting categorical to numerical features and also getting embeddings for some variables.

4.2.1.1 Train Test Split

We split the data into test and train split 30% and 70% of the data respectively. Here we treat the ratings values as the target values that need to be fit. And the rest of the columns as the input to the model

The movie year column is extracted from the title column and the title column is then dropped.

4.2.1.2 Converting categorical to numerical features

In this part categorical features are converted to numerical features like the age column which is a categorical variable is converted to a ordinal variable. The gender and occupation variable are also converted to the required format to be processed by the model of ordinal and nominal variables.

4.2.1.2 Generating embeddings

To generate the embeddings of variables which need to be used by the deep part of the model we generate the embeddings using the `torch.nn.Embedding` function which converts the one hot encoded variables to the latent space. Here for the initial setup we set the latent space size to be 100 for both the userId and the movieId

4.2.2 Setting up the neural network

In this part we define the architecture of the neural network that was used. The model was implemented using pytorch due to the ease of debugging available in the framework.

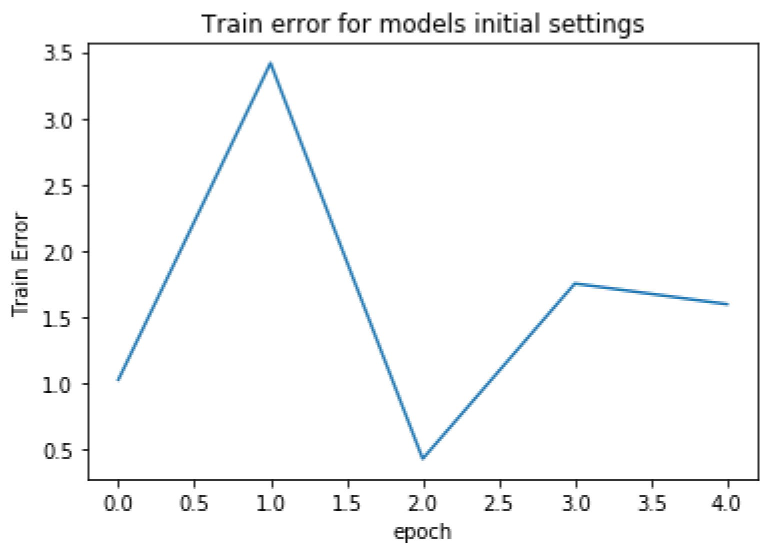
4.2.2.1 Neural network parameters

The parameters that need to be set for the Neural network are the following:

- We chose the neural network to have two hidden layers of size 100 and 50 each. These default values were chosen from the network architecture in the original paper. These was not experimented with much due to the restrictions in the compute power available.
- For the initial part we will be using dropout values of 0.5 and 0.2 for the hidden layers respectively. This will be trained further to find the optimal value for the dropouts
- We run the experiment for 5 epochs but this will be tested and tuned as well.
- For the gradient descent algorithm we have developed with the alternatives of Adam and RMSprop and the appropriate algorithm will be chose from these as well

4.2.3 Hyperparameter Tuning

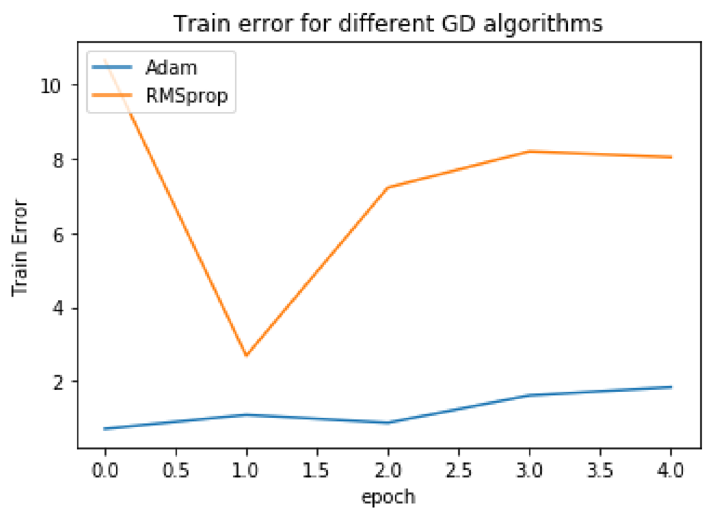
With the initial settings we see that the test error is reducing with each epoch and that the model has already converged in the first epoch



We also notice that the test mean squared error is 2.43 and the mean absolute error is 1.21 and thus the model is generalizing well.

4.2.3.1 Gradient descent algorithm

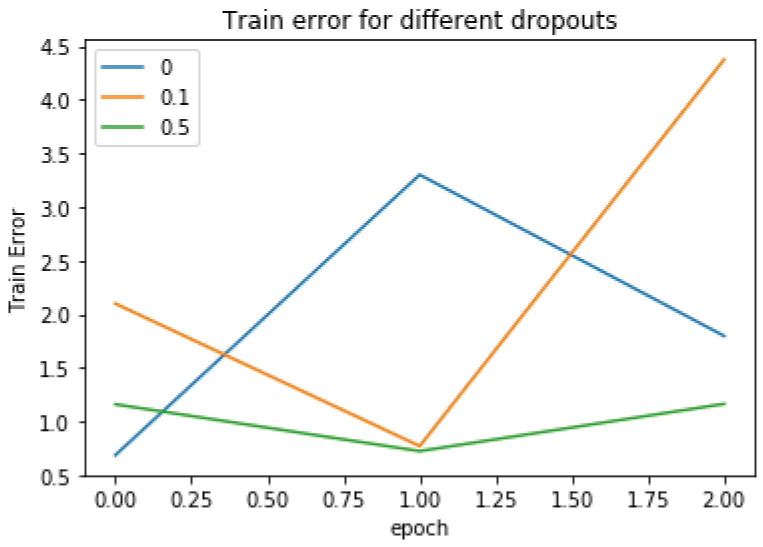
We will now train for the multiple gradient descent algorithms available to us and try to understand the the error for each of the runs



We can see that the Adam seems to clearly be performing better than RMSprop and thus we will choose adam for our implementation as even the Test MSE for Adam and RMSprop are 3.83 and 14.30 after 5 epochs and thus the adam algorithm performs better

4.2.3.1 Dropout

We choose the dropout values to be optimized from 0,0.1 and 0.5 and the best value is chosen from the methods



We can see that the model performs the best for dropout values of 0.5 and 0.5. This is further bolstered by the test MSE and MAE of 1.358 and 0.97 respectively and as these are really low we choose dropout values of 0.5 and 0.5

4.2.4 Model Evaluation

We used Mean Square Error (MSE) and Mean Absolute Error (MAE) as the evaluation error metrics of the model. The two methods are the two of the most common metrics used to measure accuracy for continuous variables.

The calculation of MSE and MAE are implemented with functions in package sklearn.

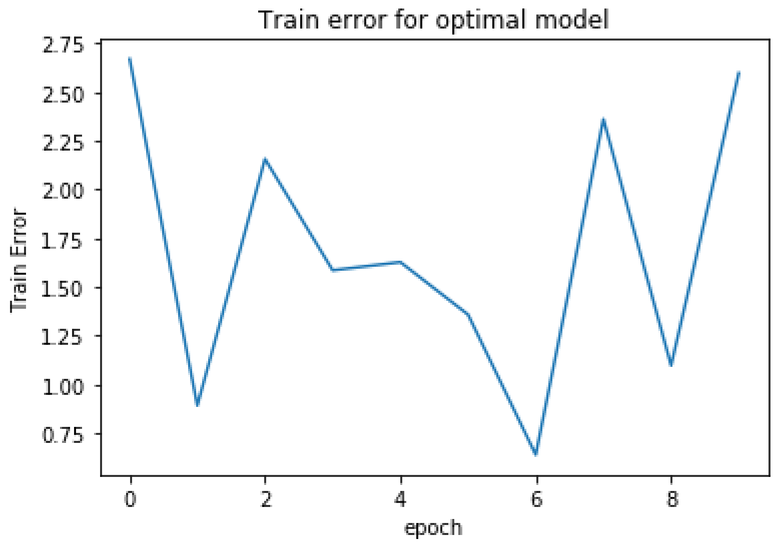
4.2.5 Benchmark Model

We used a collaborative filtering recommendation system using k-Nearest Neighbors (kNN) as our baseline model. The model is calculated using Surprise, a Python scikit library for recommender systems.

The other settings of the model are the same as our wide and deep model. We also used the 3-fold cross validation as our cross-validation splitting strategy, and we used Mean Square Error (MSE) and Mean Absolute Error (MAE) as the evaluation error metrics of the model. The lowest MAE we got from KNN-CF model is 0.73, and the lowest MSE is 0.93.

4.2.6 Model Evaluation

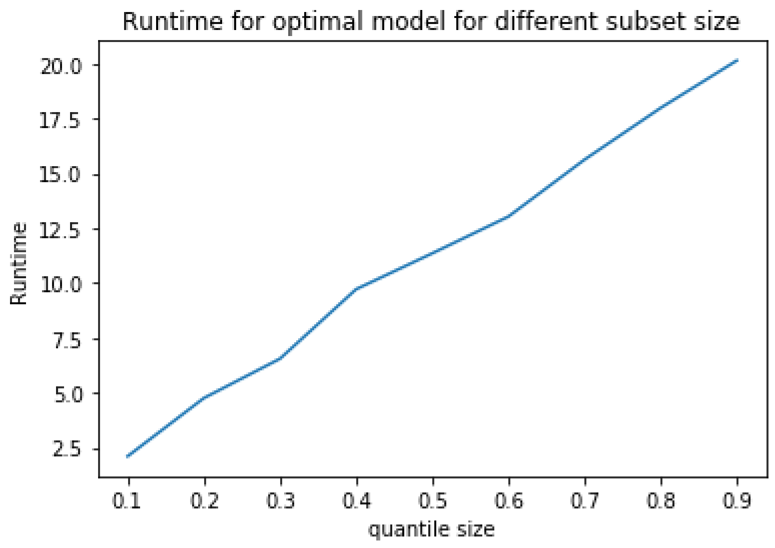
We evaluate the model with the optimal hyperparameters and try to compare it with the benchmark model.



We can see that the model converges quickly. While calculating the test statistic we see that the test mean squared error is 1.35 test mean absolute error is 0.92. We notice that the test error values are pretty low but we are still lower than the benchmark model and thus we further exploration is required to firmly establish if the method is better or worse than the benchmark as the benchmark chosen is also a good method. Due to the high standard of the benchmark these results are not very discouraging as they are still very close to the benchmark model.

4.2.6.1 Evaluation - Running Time

Diving the dataset into smaller subsets and running the algorithms we can see that the algorithm training time increases linearly with the number of users and thus the training time of the algorithm is O(n) with respect to the number of users which is good.



4.2.7 Results and Conclusion

We observe that the model seems to perform really close to the benchmark and could be further improved with further hyper parameter tuning. We notice that the algorithm does produce very low MAE and MSE for the movielens data set.

As a business the current problem is that we are performing lower than the benchmark and the Factorization machine models and also as the model is more complex it still requires more infrastructure to train and support it and thus, till be are able to tune the model to perform better than the Factorization machines and KNN it would make no sense for us as a business to use the wide and deep model for our business use case. Though it might help in some cases as the model is able to generalize well and thus could be used for more esoteric users.

The idea should still be pursued more as the paper the model is based on claims better generalization than KNN and thus we need to explore the hyperparameter space more to get the desired result