4.1 Movie Recommendations

June 6, 2019

1 Movie Recommendations

In this case study we will build multiple model for making recommendations, validate and compare them.

1.1 Identification Information

Name: Alex Perusse
MIT xPro Username: aperusse
MIT xPro Email: aperusse@cityyear.org

1.2 Data Exporation

To begin we import the data and perform some cursory data exploration.

[2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

from surprise import Dataset, SVD, NormalPredictor, BaselineOnly, KNNBasic, NMF
from surprise.model_selection import cross_validate, KFold
```

[3]:
```
ratingsdata = Dataset.load_builtin('ml-100k')
col_names = ['user_id', 'item_id', 'rating', 'timestamp']
rawdata = pd.read_table(ratingsdata.ratings_file, names=col_names)
```

```
D:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:3:
FutureWarning: read_table is deprecated, use read_csv instead, passing sep='\t'.
  This is separate from the ipykernel package so we can avoid doing imports
until
```

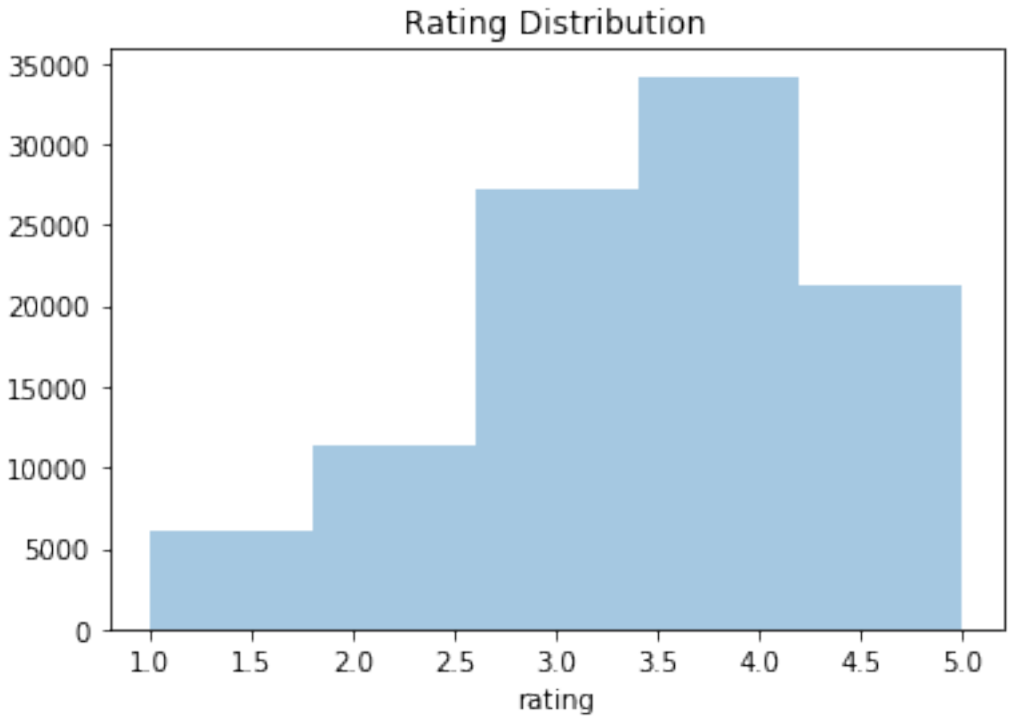
[4]:
```
rawdata.describe()
```

[4]:

	user_id	item_id	rating	timestamp
count	100000.00000	100000.000000	100000.000000	1.000000e+05
mean	462.48475	425.530130	3.529860	8.835289e+08
std	266.61442	330.798356	1.125674	5.343856e+06

```
min          1.00000       1.000000       1.000000  8.747247e+08
25%        254.00000     175.000000       3.000000  8.794487e+08
50%        447.00000     322.000000       4.000000  8.828269e+08
75%        682.00000     631.000000       4.000000  8.826000e+08
max        943.00000    1682.000000       5.000000  8.932866e+08
```

[5]:
```
sns.distplot(rawdata['rating'], bins=5, kde=False)
plt.title("Rating Distribution")
plt.show()
```



1.3 Question 1: Data Analysis

Describe the dataset. How many ratings are in the dataset? How would you describe the distribution of ratings? Is there anything else we should observe? Make sure the histogram is visible in the notebook.

In this dataset, we are given 100000 User/Item/Rating pairings, from which we hope to make recommendations. Ratings are given on a scale from 1 to 5, where 5 is the most positive rating. As you might expect, ratings are distributed such that a 4 is the most likely response followed by a 3 or a 5. 1's or 2's are rarer and thus probably carry a little more information than a 4.

1.4 Baseline Model - Random

As our baseline we will use `NormalPredictor`, which simply predicts a random rating based on the distribution of the training set, which is assumed to be normal.

[31]:
```
model_baseline = NormalPredictor()
results_baseline = cross_validate(model_baseline, ratingsdata,␣
 ↪measures=['RMSE'], cv=100, verbose=False)
m, s = results_baseline['test_rmse'].mean(), 2*results_baseline['test_rmse'].
 ↪std()
print(f"RMSE Over 5 Folds: {m} +/- {s}")
```

```
RMSE Over 5 Folds: 1.5252159899507298 +/- 0.05840238755785964
```

1.5 User-Based Collaborative Filtering

User-based Collaborative Filtering works on the premise that people who rated things similarly in the past will do so in the future. It makes predictions by averaging the ratings that similar users would have given the item.

[23]:
```
model_user = KNNBasic(sim_options={'user_based': True}, verbose=False)
results_user = cross_validate(model_user, ratingsdata, measures=['RMSE'],␣
 ↪cv=100, verbose=False)
m, s = results_user['test_rmse'].mean(), 2*results_user['test_rmse'].std()
print(f"RMSE Over 100 Folds: {m} +/- {s}")
```

```
RMSE Over 100 Folds: 0.9670531814311453 +/- 0.050096632817287025
```

1.6 Item-Based Collaborative Filtering

Similarly, item-based collaborative filtering makes predictions about items using items which have receieved similar ratings.

[4]:
```
model_item = KNNBasic(sim_options={'user_based': False}, verbose=False)
results_item = cross_validate(model_item, ratingsdata, measures=['RMSE'],␣
 ↪cv=100, verbose=False)
m, s = results_item['test_rmse'].mean(), 2*results_item['test_rmse'].std()
print(f"RMSE Over 100 Folds: {m} +/- {s}")
```

```
RMSE Over 100 Folds: 0.9607477776787791 +/- 0.0458838205226042
```

1.7 Question 2: Collaborative Filtering Models

Compare the results from the user-user and item-item models. How do they compare to each other? How do they compare to our original "random" model? Can you provide any intuition as to why the results came out the way they did?

Atleast in our run, the results weren't that different with user-based and item-based collaborative filtering with item-based only barely having an advantage, $0.961 < 0.966$. However, both were significantly better than the random model whose RMSE was considerably higher, $0.966 < 1.519$, than either of the collaborative filtering based approahces. This makes sense of course, because the random results are... random. It gives a nice sense of what would happen if we simply rolled a die to make recommendations as opposed to using the data we have.

1.8 Model 4: Matrix Factorization

[13]:
```
model_svd = SVD()
folds = 50
results_item = cross_validate(model_svd, ratingsdata, measures=['RMSE'],␣
 ↪cv=folds, verbose=False)
m, s = results_item['test_rmse'].mean(), 2*results_item['test_rmse'].std()
print(f"RMSE Over {folds} Folds: {m} +/- {s}")
```

```
RMSE Over 50 Folds: 0.9242514125226332 +/- 0.031770023605667906
```

1.9 Question 3: Matrix Factorization Model

The matrix factorization model is different from the collaborative filtering models. Briefly describe this difference. Also, compare the RMSE again. Does it improve? Can you offer any reasoning as to why that might be?

Where Collaborative filtering takes advantage of nearest neighbor to try to make recomendations to user based on similar users or items based on similar items, matrix factorization is useful when you believe there are latent factors, that is features not directly represented in the data, that are truly the levers in someones preferences. For example, perhaps there are certain actors or actressses who appears in movies which cause them to be popular. Matrix Factorization allows one to model this possiblity and try to derive that matrix of latent factors.

We in fact did see a improvement in performance by using matrix factorication by a small margin, $0.924 < 0.961$.

1.10 Taking Model Criticism Further: Precision and Recall

Going beyond the RMSE, we can evaluate the precision and recall of our recommender as well in a similar fachion to how one does in a classification problem.

[14]:
```
def precision_recall_at_k(predictions, k=10, threshold=3.5):
    '''Return precision and recall at k metrics for each user.'''

    # First map the predictions to each user.
    user_est_true = dict()
    for uid, _, true_r, est, _ in predictions:
        current = user_est_true.get(uid, list())
        current.append((est, true_r))
        user_est_true[uid] = current

    precisions = dict()
    recalls = dict()
    for uid, user_ratings in user_est_true.items():

        # Sort user ratings by estimated value
        user_ratings.sort(key=lambda x: x[0], reverse=True)

        # Number of relevant items
        n_rel = sum((true_r >= threshold) for (_, true_r) in user_ratings)
```

```
        # Number of recommended items in top k
        n_rec_k = sum((est >= threshold) for (est, _) in user_ratings[:k])

        # Number of relevant and recommended items in top k
        n_rel_and_rec_k = sum(((true_r >= threshold) and (est >= threshold))
                              for (est, true_r) in user_ratings[:k])

        # Precision@K: Proportion of recommended items that are relevant
        precisions[uid] = n_rel_and_rec_k / n_rec_k if n_rec_k != 0 else 1

        # Recall@K: Proportion of relevant items that are recommended
        recalls[uid] = n_rel_and_rec_k / n_rel if n_rel != 0 else 1

    return precisions, recalls
```

[25]:
```
algos = {'baseline': model_baseline, 'user': model_user, 'item': model_item,␣
 ↪'SVD': model_svd}

for name, algo in algos.items():
    print(name)
    p_arr = []
    r_arr = []
    for folds in [5, 10]:
        print(f"Over {folds}-folds")
        kf = KFold(n_splits=5)
        fold = 0
        for trainset, testset in kf.split(ratingsdata):
            algo.fit(trainset)
            predictions = algo.test(testset)
            precisions, recalls = precision_recall_at_k(predictions, k=5,␣
 ↪threshold=4)
            fold += 1

            # Precision and recall can then be averaged over all users
            #print(f"Fold {fold}")
            p = sum(prec for prec in precisions.values()) / len(precisions)
            r = sum(rec for rec in recalls.values()) / len(recalls)
            p_arr.append(p)
            r_arr.append(r)
        print(f"Average Precision: {np.mean(p_arr)}")
        print(f"Average Recall: {np.mean(r_arr)}")
```

```
baseline
Over 5-folds
Average Precision: 0.6012145022718249
Average Recall: 0.2707458905346869
Over 10-folds
```

```
Average Precision: 0.6024572031772 72
Average Recall: 0.2709060915008587
user
Over 5-folds
Average Precision: 0.8443762436192029
Average Recall: 0.27138036262948206
Over 10-folds
Average Precision: 0.8457641624272216
Average Recall: 0.27219540289037625
item
Over 5-folds
Average Precision: 0.9425407266620075
Average Recall: 0.17057555577493896
Over 10-folds
Average Precision: 0.9428893250490159
Average Recall: 0.17027643050263264
SVD
Over 5-folds
Average Precision: 0.8733167399522237
Average Recall: 0.26416978079592945
Over 10-folds
Average Precision: 0.8723540686805373
Average Recall: 0.2611656280625076
```

1.11 Question 4: Precision/Recall

Compute the precision and recall, for each of the 4 models, at k = 5 and 10. This is 2 x 2 x 4 = 16 numerical values. Do you note anything interesting about these values? Anything differerent from the RMSE values you computed above?

As you would expect, the random model performs the worst in terms of precision of all of the models, although not worse in Recall. That brings us to the user and item collaborative filters. Here we see a class tradeoff in precision and recall. Item-based collaborative filtering gets the higher precision but at the cost of having the worst recall. This is likely because the item-based model returns a lot of negative recommendations and few positives for any user. As a result, there are fewer false positives because there are also fewer positives identified by the model at all. You might interpret this as more conservative behavior, only recommending what you like.

The SVD plays the balance the best, with a higher precision than user-based and with nearly the same recall.

From our analysis thus far, I would continue to conlude that matrix factorization seems to be the best model that we've tried.

[29]:
```
def get_top_n(predictions, n=5):
    '''Return the top-N recommendation for each user from a set of predictions.

    Args:
        predictions(list of Prediction objects): The list of predictions, as
            returned by the test method of an algorithm.
        n(int): The number of recommendation to output for each user. Default
```

```
            is 10.

    Returns:
    A dict where keys are user (raw) ids and values are lists of tuples:
        [(raw item id, rating estimation), ...] of size n.
    '''

    # First map the predictions to each user.
    top_n = dict()
    for uid, iid, true_r, est, _ in predictions:
        current = top_n.get(uid, [])
        current.append((iid, est))
        top_n[uid] = current

    # Then sort the predictions for each user and retrieve the k highest ones.
    for uid, user_ratings in top_n.items():
        user_ratings.sort(key=lambda x: x[1], reverse=True)
        top_n[uid] = user_ratings[:n]

    return top_n
```

[52]:
```
trainset = ratingsdata.build_full_trainset()
model_baseline.fit(trainset)
predictions = model_baseline.test(trainset.build_anti_testset())
topn = get_top_n(predictions, n=5)
```

[58]:
```
# Predictions for user `1`
topn['1']
```

[58]:
```
[('768', 5), ('277', 5), ('526', 5), ('512', 5), ('480', 5)]
```

1.12 Question 5: Top N Predicions

Do the top n predictions that you received make sense? What is the rating value (1-5) of these predictions? How could you use these predictions in the real-world if you were trying to build a generic content recommender system for a company?

Scanning through the predictions our model is returning 5 recommended items all with a predicted rating of 5. This is exactly what I would expect, since it should be the five highest rated items for that individual.

In order to leverage this type of system in production for a real-world company my strategy would be to generate a much large set of potential recommendation for each user, say the top 100, and then randomly select a subset of those to actually recommend to the user. This would help ensure that user gets different recommendations each time they come to the site, which helps support exploration.

[ ]:
```

```