Recommendation report

2015004248
Jo Hyung Jung

1. Overview

In this project, I predict ratings of movies in test data by using training data containing movie ratings of users. There are typically two ways to recommend, content-based method and collaborative filtering method respectively.

I used collaborative filtering method. Since there are several methods in collaborative filtering, I had to implement those methods and compare each of them. I found that using ‘user-based collaborative filtering’ is the best way based on RMSE value.

2. Run-time environment

- OS: Ubuntu 18.04
- Language: Python 3.6
- Libraries:
    Python argparse module: for getting command-line arguments.

3. Summary of algorithm

I tested several methods in collaborative filtering to find out best result. Following is the each of them respectively.

1. Rating matrix

i) User-based collaborative filtering
Use relationship of users to build rating matrix.

ii) Item-based collaborative filtering
User relationship of items to build item matrix. In this project, the information of relationship of items is much less than of users. So I used ‘User-based collaborative filtering’.

2. Similarity measure

i) Cosine similarity
w(a,i) = sum[{v(a,j) / sqrt{sum(v(a,k)**2}} * {v(i,j) / sqrt{sum(v(i,k)**2}}]

ii) Pearson coefficient correlation similarity
w(a,i) = sum[{v(a,j)-mean of v(a)} * {v(i,j)-mean of v(i)}] /
sqrt[sum{(v(a,j)-mean of v(a))**2} * sum{(v(i,j)-mean of v(i))**2}]

3. Aggregation of ratings

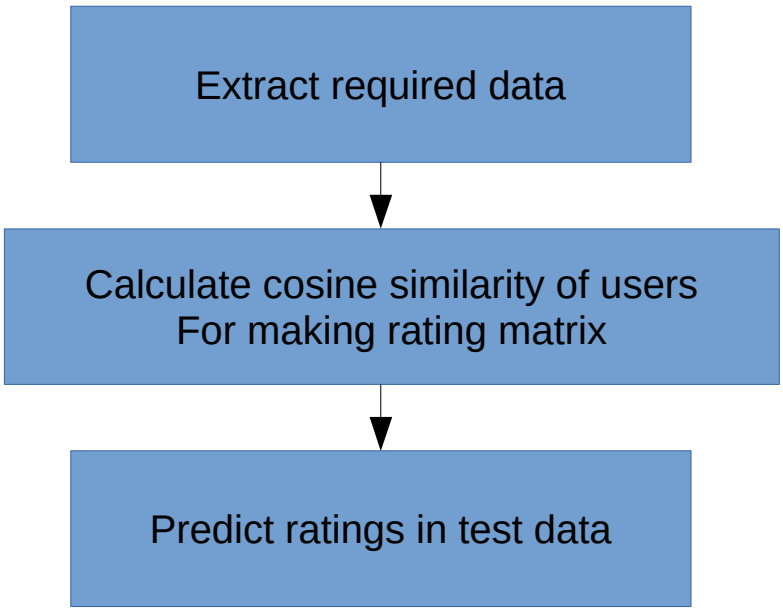
i) Simple average
r(c,s) = sum{r(c`,s)} / N

ii) Weighted average
r(c,s) = sum{sim(c,c`) * r(c`,s)} * k

iii) Weighted average (normalized version)
r(c,s) = mean of r(c) + sum[sim(c,c`) * {r(c`,s) -mean of r(c`)}] * k

4. Implementation details

In testing above methods, I found out using ‘User-based’ & ‘Cosine similarity’ & ‘Weighted average (normalized version)’ is the best solution. Below is the flow chart.



[Flow chart]

5. Analysis of code

i) Data structures

I used one class named ‘Recommender’

```
class Recommender:
    def __init__(self, args):
        # Save output file name
        self.output = args.training_file + "_prediction.txt"

        # Training data(User): {'user id': {'item id': rating ...} ...}
        # Training data(Item): {'item id': {'user id': rating ...} ...}
        # User average rating: {'user id': average value ...}
        # Item average rating: {'item id': average value ...}
        self.training_user = {}
        self.training_item = {}
        self.user_avg = {}
        self.item_avg = {}

        # Use user based only
        self.metric = 1
```

```
        # Test data: {'user id': {'item id': rating ...} ...}
        self.test = {}
        with open(args.test_file, 'r') as f:
            for line in f.readlines():
                user_id, item_id, _, _ = map(int,line.split())
                if user_id not in self.test:
                    self.test[user_id] = {item_id: 0}
                else:
                    self.test[user_id][item_id] = 0

        # Rating matrix(User): {'user id': {'user id': similarity ...} ...}
        self.user_matrix = {i: {j: 0.0 for j in self.training_user.keys()} for i in self.training
_user.keys()}

        # In case of using item based
        if self.metric != 1:
            # Rating matrix(Item): {'item id': {'item id': similarity ...} ...}
            self.item_matrix = {i: {j: 0.0 for j in self.training_item.keys()} for i in self.trai
ning_item.keys()}
```

There are 9 attributes in ‘Recommender’ class.

1. output: represents output file name
2. training_user: represents training data in user-based model
3. training_item: represents training data in item-based model
4. user_avg: represents rating average of each user in user-based model
5. item_avg: represents rating average of each item in item-based model
6. metric: represents whether using user-based model or item-based model. Fixed to use user-based model.
7. test: represents test data (Target user/item to predict rating)
8. user_matrix: represents rating matrix in user-based model
9. item_matrix: represents rating matrix in item-based model

ii) Functions

I use 9 functions in ‘Recommender’ class.

- __init__

```
class Recommender:
    def __init__(self, args):
        # Save output file name
        self.output = args.training_file + "_prediction.txt"

        # Training data(User): {'user id': {'item id': rating ...} ...}
        # Training data(Item): {'item id': {'user id': rating ...} ...}
        # User average rating: {'user id': average value ...}
        # Item average rating: {'item id': average value ...}
        self.training_user = {}
        self.training_item = {}
        self.user_avg = {}
        self.item_avg = {}

        # Use user based only
        self.metric = 1

        with open(args.training_file, 'r') as f:
            for line in f.readlines():
                user_id, item_id, rating, _ = map(int,line.split())
                # User
                if user_id not in self.training_user:
                    self.training_user[user_id] = {item_id: rating}
                else:
                    self.training_user[user_id][item_id] = rating

                # User avg rating
                if user_id not in self.user_avg:
                    self.user_avg[user_id] = [rating]
                else:
                    self.user_avg[user_id].append(rating)

                # In case of using item based
                if self.metric != 1:
                    # Item
                    if item_id not in self.training_item:
                        self.training_item[item_id] = {user_id: rating}
                    else:
                        self.training_item[item_id][user_id] = rating

                    # Item avg rating
                    if item_id not in self.item_avg:
                        self.item_avg[item_id] = [rating]
                    else:
                        self.item_avg[item_id].append(rating)

        # Process user average value
        for uid in self.user_avg.keys():
            self.user_avg[uid] = sum(self.user_avg[uid]) / len(self.user_avg[uid])

        # In case of using item based
        if self.metric != 1:
            # Process item average value
            for iid in self.item_avg.keys():
                self.item_avg[iid] = sum(self.item_avg[iid]) / len(self.item_avg[iid])

        # Test data: {'user id': {'item id': rating ...} ...}
        self.test = {}
        with open(args.test_file, 'r') as f:
            for line in f.readlines():
                user_id, item_id, _, _ = map(int,line.split())
                if user_id not in self.test:
                    self.test[user_id] = {item_id: 0}
                else:
                    self.test[user_id][item_id] = 0

        # Rating matrix(User): {'user id': {'user id': similarity ...} ...}
        self.user_matrix = {i: {j: 0.0 for j in self.training_user.keys()} for i in self.training
_user.keys()}

        # In case of using item based
        if self.metric != 1:
            # Rating matrix(Item): {'item id': {'item id': similarity ...} ...}
            self.item_matrix = {i: {j: 0.0 for j in self.training_item.keys()} for i in self.trai
ning_item.keys()}

        # Get rating matrix(User) values
        self._get_user_matrix()

        # In case of using item based
        if self.metric != 1:
            # Get rating matrix(Item) values
            self._get_item_matrix()

        # Predict: weighted average
        self._predict()

        # Save prediction data
        self._write()
```

Feature: Initialize attributes and predict ratings.
Arguments: args → python 'Argument Parser' object
Return value: None

In this function, attributes are initialized and total process is proceeded.

First, initialize variables and set metric value. Metric value is set to 1 denoting 'User based model' which is the best solution.

Second, read training file and extract data. I don't use timeline data in this project. Training data is saved as dictionary **format: {'user id': {'item id': rating …} …}**.
After processing training data, calculate average rating values of each user. It is also saved as dictionary **format: {'user id': average value …}**.

Third, read test file and extract data. Target user and item id is saved as dictionary **format: {'user id': {'item id': rating …} …}**. Then make initial rating matrix.

Fourth, make rating matrix using function **'_get_user_matrix'**. More specific information of this function is below.

Fifth, make prediction using rating matrix above. Then write result in appropriate file format.

-_get_cosine_similarity

```
def _get_cosine_similarity(self, v1, v2):
    ret = 0.0
    v1_size = 0.0
    v2_size = 0.0
    # Assume v1 has same length as v2
    for i in range(len(v1)):
        ret += v1[i]*v2[i]
        v1_size += v1[i]**2
        v2_size += v2[i]**2

    return ret / (v1_size**(1/2) * v2_size**(1/2))
```

Feature: Calculate cosine similarity
Arguments: v1 → list, v2 → list
Return value: ret / (v1_size(1/2) * v2_size**(1/2))**

In this function, calculate cosine similarity of two given vectors. Same size of two vectors is guaranteed. To avoid divide zero error, caller of _get_cosine_similarity checks whether v1 and v2 are empty or not.

-_get_pcc_similarity

```
def _get_pcc_similarity(self, v1, v2):
    ret = 0.0
    v1_size = 0.0
    v2_size = 0.0
    v1_mean = sum(v1) / len(v1)
    v2_mean = sum(v2) / len(v2)
    # Assume v1 has same length as v2
    for i in range(len(v1)):
        ret += (v1[i]-v1_mean)*(v2[i]-v2_mean)
        v1_size += (v1[i]-v1_mean)**2
        v2_size += (v2[i]-v2_mean)**2

    try:
        return ret / (v1_size**(1/2) * v2_size**(1/2))
    except ZeroDivisionError:
        return 0.0
```

Feature: Calculate pearson correlation coefficient similarity
Arguments: v1→ list, v2 → list
Return value: ret / (v1_size(1/2) * v2_size**(1/2)) or 0.0**

In this function, calculate pearson correlation coefficient similarity of two given vectors. Same size of two vectors is guaranteed. If numerator is zero, return 0.0 to avoid 'ZeroDivisionError'. This function isn't used for better result.

-_get_user_matrix

```
def _get_user_matrix(self):
    # Cosine similarity method
    # Set threshold: 0%
    threshold = len(self.training_user.keys()) * 0.0
    for i in self.training_user.keys():
        for j in self.training_user.keys():
            if i == j:
                self.user_matrix[i][j] = 1.0
            else:
                v1_dict = self.training_user[i]
                v2_dict = self.training_user[j]
                v1 = []
                v2 = []
                for iid in set(list(v1_dict.keys())+list(v2_dict.keys())):
                    if iid in v1_dict and iid in v2_dict:
                        v1.append(v1_dict[iid])
                        v2.append(v2_dict[iid])

                # Avoid empty list
                if v1 and v2:
                    if len(v1) >= threshold:
                        self.user_matrix[i][j] = self._get_cosine_similarity(v1,v2)
                    else:
                        self.user_matrix[i][j] = 0.0
```

Feature: Create rating matrix in user-based method.
Arguments: None
Return value: None

In this function, create rating matrix in user-based method. By iterating every user, calculate similarity of two given users. Cosine similarity is used to calculate similarity. If there are no common rated items, set similarity to zero. This function isn't used for better result.

-_get_item_matrix

```
def _get_item_matrix(self):
    # Cosine similarity method
    for i in self.training_item.keys():
        for j in self.training_item.keys():
            if i == j:
                self.item_matrix[i][j] = 1.0
            else:
                v1_dict = self.training_item[i]
                v2_dict = self.training_item[j]
                v1 = []
                v2 = []
                for uid in set(list(v1_dict.keys())+list(v2_dict.keys())):
                    if uid in v1_dict and uid in v2_dict:
                        v1.append(v1_dict[uid])
                        v2.append(v2_dict[uid])

                # Avoid empty list
                if v1 and v2:
                    self.item_matrix[i][j] = self._get_cosine_similarity(v1,v2)
```

Feature: Create rating matrix in item-based method.
Arguments: None
Return value: None

In this function, create rating matrix in item-based method. By iterating every user, calculate similarity of two given users. Cosine similarity is used to calculate similarity. If there are no common values, set similarity to zero.

-_get_rating_user

```
def _get_rating_user(self, uid, iid):
    fraction = 0.0
    denominator = 0.0

    for user in self.user_matrix.keys():
        if iid in self.training_user[user]:
            fraction += self.user_matrix[uid][user] * (self.training_user[user][iid]-self.use
r_avg[user])
            denominator += self.user_matrix[uid][user]

    # Avoid divide by zero
    try:
        ret = self.user_avg[uid] + (fraction / denominator)
    except ZeroDivisionError:
        ret = self.user_avg[uid]

    ret = int(ret+0.5)
    # Clip value
    if ret > 5:
        ret = 5
    if ret < 1:
        ret = 1
    return ret
```

Feature: predict rating in user-based method
Arguments: uid→ int, iid→ int
Return value: ret→ int

In this function, rating is predicted by using rating matrix and given user id & item id. I used weighted average aggregation with normalization. Normalization is done by subtracting mean values. If 'ZeroDivisionError' occurs, mean value of given user is returned. Return value is clipped when it's over 5 (Maximum rating value) or below 1 (Minimum rating value).

-_get_rating_item

```
    def _get_rating_item(self, uid, iid):
        fraction = 0.0
        denominator = 0.0

        for item in self.item_matrix.keys():
            if item == iid:
                continue
            if uid in self.training_item[item] and iid in self.item_matrix[item]:
                fraction += self.item_matrix[item][iid] * (self.training_item[item][uid]-self.ite
m_avg[item])
                denominator += self.item_matrix[iid][item]

        # Avoid divide by zero
        try:
            ret = self.item_avg[iid] + fraction / denominator
        except ZeroDivisionError:
        # Inject lowest value
            ret = self.item_avg[iid]

        ret = int(ret+0.5)
        # Clip value
        if ret > 5:
            ret = 5
        if ret < 1:
            ret = 1
```

Feature: predict rating in item-based method
Arguments: uid→ int, iid→ int
Return value: ret→ int

In this function, rating is predicted by using rating matrix and given user id & item id. I used weighted average aggregation with normalization. Normalization is done by subtracting mean values. If 'ZeroDivisionError' occurs, mean value of given item is returned. Return value is clipped when it's over 5 (Maximum rating value) or below 1 (Minimum rating value).

-_predict

```
def _predict(self):
    # User user based only
    for uid in self.test.keys():
        for iid in self.test[uid].keys():
            if self.metric != 1:
                self.test[uid][iid] = self._get_rating_item(uid, iid)
            else:
                self.test[uid][iid] = self._get_rating_user(uid, iid)
```

Feature: Predict target ratings
Arguments: None
Return value: None

In this function, ratings in test file is predicted. Using **'_get_rating_user'** function, each of rating in user & item is predicted.

-_write

```
def _write(self):
    with open(self.output, 'w') as f:
        for uid in self.test.keys():
            for iid in self.test[uid]:
                f.write("{}\t{}\t{}\n".format(uid,iid,self.test[uid][iid]))
```

Feature: Write result values in to output file
Arguments: None
Return value: None

In this function, result values are written in to output file. Each line has given format.

iii) Code flow

```
if __name__ == "__main__":
    # Parse arguments
    parser = argparse.ArgumentParser()
    parser.add_argument("training_file", type=str)
    parser.add_argument("test_file", type=str)
    args = parser.parse_args()
    recommender = Recommender(args)
```

[In main routine]

1. Parse command-line arguments.
2. Construct 'Recommender' class

```
class Recommender:
    def __init__(self, args):
        # Save output file name
        self.output = args.training_file + "_prediction.txt"

        # Training data(User): {'user id': {'item id': rating ...} ...}
        # Training data(Item): {'item id': {'user id': rating ...} ...}
        # User average rating: {'user id': average value ...}
        # Item average rating: {'item id': average value ...}
        self.training_user = {}
        self.training_item = {}
        self.user_avg = {}
        self.item_avg = {}

        # Use user based only
        self.metric = 1

        with open(args.training_file, 'r') as f:
            for line in f.readlines():
                user_id, item_id, rating, _ = map(int,line.split())
                # User
                if user_id not in self.training_user:
                    self.training_user[user_id] = {item_id: rating}
                else:
                    self.training_user[user_id][item_id] = rating

                # User avg rating
                if user_id not in self.user_avg:
                    self.user_avg[user_id] = [rating]
                else:
                    self.user_avg[user_id].append(rating)

                # In case of using item based
                if self.metric != 1:
                    # Item
                    if item_id not in self.training_item:
                        self.training_item[item_id] = {user_id: rating}
                    else:
                        self.training_item[item_id][user_id] = rating

                    # Item avg rating
                    if item_id not in self.item_avg:
                        self.item_avg[item_id] = [rating]
                    else:
                        self.item_avg[item_id].append(rating)

        # Process user average value
        for uid in self.user_avg.keys():
            self.user_avg[uid] = sum(self.user_avg[uid]) / len(self.user_avg[uid])

        # In case of using item based
        if self.metric != 1:
            # Process item average value
            for iid in self.item_avg.keys():
                self.item_avg[iid] = sum(self.item_avg[iid]) / len(self.item_avg[iid])

        # Test data: {'user id': {'item id': rating ...} ...}
        self.test = {}
        with open(args.test_file, 'r') as f:
            for line in f.readlines():
                user_id, item_id, _, _ = map(int,line.split())
                if user_id not in self.test:
                    self.test[user_id] = {item_id: 0}
                else:
                    self.test[user_id][item_id] = 0

        # Rating matrix(User): {'user id': {'user id': similarity ...} ...}
        self.user_matrix = {i: {j: 0.0 for j in self.training_user.keys()} for i in self.training
_user.keys()}

        # In case of using item based
        if self.metric != 1:
            # Rating matrix(Item): {'item id': {'item id': similarity ...} ...}
            self.item_matrix = {i: {j: 0.0 for j in self.training_item.keys()} for i in self.trai
ning_item.keys()}

        # Get rating matrix(User) values
        self._get_user_matrix()

        # In case of using item based
        if self.metric != 1:
            # Get rating matrix(Item) values
            self._get_item_matrix()

        # Predict: weighted average
        self._predict()

        # Save prediction data
        self._write()
```

[In __init__ function]

1. Save arguments
2. Extract data from training & test file
3. Create rating matrix
4. Predict target ratings
5. Write result values in to output file

6. Instructions for execution

```
jsense@jsense:~/2020_ITE4005_2015004248/project_recommend$ python3 recommender.py u1.base u1.test
jsense@jsense:~/2020_ITE4005_2015004248/project_recommend$ python3 recommender.py u2.base u2.test
jsense@jsense:~/2020_ITE4005_2015004248/project_recommend$ python3 recommender.py u3.base u3.test
jsense@jsense:~/2020_ITE4005_2015004248/project_recommend$ python3 recommender.py u4.base u4.test
jsense@jsense:~/2020_ITE4005_2015004248/project_recommend$ python3 recommender.py u5.base u5.test
```

Type ‘python3 recommender.py “training data name” “test data name”’in terminal

7. Test result

```
jsense@jsense:~/2020_ITE4005_2015004248/project_recommend$ wine PA4.exe u1
the number of ratings that didn't be predicted: 0
the number of ratings that were unproperly predicted [ex. >=10, <0, NaN, or format errors]: 0
If the counted number is large, please check your codes again.

The bigger value means that the ratings are predicted more incorrectly
RMSE: 1.014914
jsense@jsense:~/2020_ITE4005_2015004248/project_recommend$ wine PA4.exe u2
the number of ratings that didn't be predicted: 0
the number of ratings that were unproperly predicted [ex. >=10, <0, NaN, or format errors]: 0
If the counted number is large, please check your codes again.

The bigger value means that the ratings are predicted more incorrectly
RMSE: 1.004042
jsense@jsense:~/2020_ITE4005_2015004248/project_recommend$ wine PA4.exe u3
the number of ratings that didn't be predicted: 0
the number of ratings that were unproperly predicted [ex. >=10, <0, NaN, or format errors]: 0
If the counted number is large, please check your codes again.

The bigger value means that the ratings are predicted more incorrectly
RMSE: 1.002597
jsense@jsense:~/2020_ITE4005_2015004248/project_recommend$ wine PA4.exe u4
the number of ratings that didn't be predicted: 0
the number of ratings that were unproperly predicted [ex. >=10, <0, NaN, or format errors]: 0
If the counted number is large, please check your codes again.

The bigger value means that the ratings are predicted more incorrectly
RMSE: 0.9988994
jsense@jsense:~/2020_ITE4005_2015004248/project_recommend$ wine PA4.exe u5
the number of ratings that didn't be predicted: 0
the number of ratings that were unproperly predicted [ex. >=10, <0, NaN, or format errors]: 0
If the counted number is large, please check your codes again.

The bigger value means that the ratings are predicted more incorrectly
RMSE: 0.9958665
jsense@jsense:~/2020_ITE4005_2015004248/project_recommend$
```

[Test result in Ubuntu]

8. Reference

[1] Jiawei Han. Data mining: concepts and techniques(3rd edition), 2000.