recsys

October 15, 2018

1 Sistemas de recomendação no Turismo

1.1 Filtragem colaborativa

Sistemas de recomendação baseados em colaboração consistem na recomendação de itens que pessoas com gosto semelhante preferiram no passado. Um exemplo bem explicativo é o seguinte: "Se um usuário gostou de laranjas e de maçãs, um outro usuário que gostou de laranjas também pode gostar de maçãs".

Esse tipo de sistema de recomendação tem mais aplicações práticas, como o caso da Amazon, por exemplo e evita o problema da sugestão de item repetitivos. A principal desvantagem é a exigência de mais informações dos usuários para funcionar mais precisamente. Existem dois tipos principais de filtragem colaborativa: baseada em usuário e baseada em item.

1.1.1 Filtragem colaborativa baseada em usuário

É um algoritmo baseado em memória que calcula uma determinada pontuação de similaridade entre os usuários. Com base nessa pontuação, ele seleciona os usuários mais semelhantes e recomenda produtos que esses usuários semelhantes tenham gostado anteriormente.

Como pode ser visto na foto 1, Diego gostou de todos os pratos que o Caio gostou (camarão e carne) e de mais outros dois (sopa e salada). Considerando que Caio e Diego possuem gostos semelhantes, pela filtragem colaborativa baseada em usuários, seria recomendado para o Caio os outros dois pratos que Diego comeu: sopa e salada.

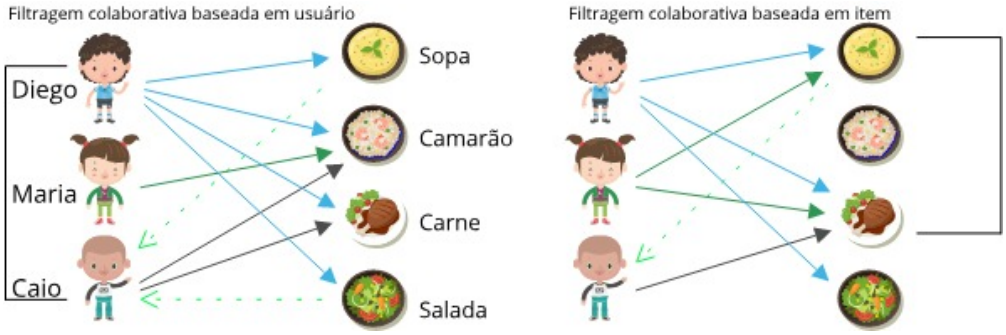

Foto 1 - Comparativo entre user-based e item-based collaborative filtering

1.1.2 Filtragem colaborativa baseada em item

É um algoritmo baseado em modelo para fazer recomendações. Nesse algoritmo são calculadas as semelhanças entre cada par de itens, esses valores de similaridade são usados para prever classificações para pares de itens de um usuário Y que não estão presentes no conjunto de dados.

Por outro viés, voltando a olhar a foto 1, analisando a carne e a sopa, podemos ver eles foram desgustados e aprovados pela Maria e pelo Diego, logo, podemos dizer que esses pratos são "semelhantes". Dessa forma, por exemplo, como o Caio gostou da carne, pela filtragem colaborativa baseada em itens, seria recomendado para a ele um prato de sopa.

1.2 Sistemas de recomendação baseados em modelo

Em sistemas de recomendação baseados em modelo, as informações são extraídas da base de dados original para construir um "exemplar" dessa base, ou seja, um **modelo** dessa base original.

Vantagens:

- Escalabilidade
- Velocidade na recomendação
- Menor sobreajuste (*overfitting*)

Desvantagens:

- É mais inflexível
- Dependendo da construção do modelo, a qualidade das previsões pode ser menor.

1.3 Sistemas baseados em memória

Em sistemas de recomendação baseados em memória, toda a base de dados é usada para calcular a similaridade e recomendar um item.

Vantagens:

- Possui uma implementação mais simples
- As recomendações geralmente possuem mais qualidade

Desvantagens:

- É extramamente lento
- Consome muitos recursos, como mémoria RAM
- Pode adquirir sobreajuste (*overfitting*)

1.4 Critérios de escolha do tipo de algoritmo baseado em colaboração

Antes de explicar o resto dessa seção, vale mencionar que não estão sendo considerados algoritmos de sistemas de recomendação baseados em conteúdo por dois motivos principais:

1. O principal problema dos sistemas baseado em conteúdo, recomendação "repetitiva" pesa muito no contexto de turismo. Se um turista já visitou três praias, por exemplo, não é interessante que seja recomendado para ele outra praia.
2. Como evidência empírica, provou-se que os sitemas de recomendação baseados em colaboração garantem uma melhor personalização, agradando mais aos usuários.

1.4.1 Velocidade e escalabilidade

O ramo do turismo é dinâmico. É melhor que turistas passem mais tempo se divertindo do que esperando mais tempo por uma recomendação. Sendo assim, é importante que o sistema faça recomendações para usuários quase instataneamente. Atrelado a isso, é importante citar que a base de dados pode crescer bastante, sendo assim é importante garantir a escalabilidade para que o algoritmo recomende rapidamente mesmo que a base de dados cresça muito.

1.4.2 Qualidade das previsões

É necessário que o algoritmo faça boas recomendações sem deixar de lado o desempenho do mesmo.

1.4.3 Fácil atualização

O conjunto de dados é atualizado constantamente devido a coleta de avaliações. Sendo assim, o algoritmo deve manipular essa informação atualizada rapidamente.

1.5 Tipo de algoritmo escolhido

Após a análise das vantagens e desvantages, resolvemos utilizar algoritmos baseados em modelo, mais especificamente, foi escolhido utilizar algoritmos de filtragem colaborativa baseada em itens.

1.6 Métricas de avaliação offline

1.6.1 *Precision*

De todos os itens recomendados, quantos deles o usuário realmente gostou?

A seguinte formulada é usada para o cálculo da *precision*:

$$P = \frac{tp}{tp + fp}$$

Onde tp é o número de itens recomendados que o usuário gostou e $tp + fp$ é o número total de itens recomendados.

Vamos supor que dez itens tenham sido recomendados para o usuário e desses dez itens ele tenha gostado de 8, o cálculo da *precision* seria da seguinte forma:

$$P = \frac{tp}{tp + fp} = \frac{8}{10} = 0.8$$

1.6.2 *Recall*

Qual a proporção de itens que um usuário gosta foi realmente recomendada?

A seguinte fórmula é usada para calcular o *recall*:

$$R = \frac{tp}{tp + fn}$$

Onde tp é o número de itens recomendados que o usuário gosta e $tp + fn$ é o número total de itens que usuário gosta. Vamos supor que um usuário gosta de dez itens e o sistema recomenda para o usuário sete desses dez itens, o cálculo do *recall* seria da seguinte forma:

$$R = \frac{tp}{tp + fn} = \frac{7}{10} = 0.7$$

O objetivo nesses casos é maximizar tanto o *recall* como a *precision*. Quanto maior o valor de ambos, melhor.

1.6.3 *Root Mean Squared Error* (RMSE)

Calcula o erro das avaliações estimadas. A seguinte fórmula é usada para o cálculo do RMSE:

$$\text{RMSE} = \sqrt{\frac{1}{|\hat{R}|} \sum_{\hat{r}_{ui} \in \hat{R}} (r_{ui} - \hat{r}_{ui})^2}.$$

Por exemplo, se um usuário avaliou um item com nota cinco e o sistema previu a nota quatro, o RMSE é um.

Sendo assim, quanto menor o RMSE, melhor.

1.7 Base de dados

Foi utilizada a base de dados real de turismo com 31361 usuários, 92 locais e 76071 avaliações. Mais informações da mesma pode ser visto abaixo:

```
In [2]: import random
        import numpy as np
        import pandas as pd

        data_v = pd.read_csv('tourismn.csv', names=['usuário', 'item', 'avaliação'])
        data_v.sort_values(by=['usuário'], ascending=False).head()
```

Out[2]:

	usuário	item	avaliação
9645	31361	92	4
1716	31360	45	5
19704	31360	87	5
25227	31359	87	3
5245	31359	45	5

```
In [3]: data_v.sort_values(by=['item'], ascending=False).head()
```

Out[3]:

	usuário	item	avaliação
7186	16224	92	3
7957	16027	92	4
7950	27961	92	5
7951	15708	92	5
7952	7412	92	5

```
In [4]: data_v.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 76071 entries, 0 to 76070
Data columns (total 3 columns):
usuário      76071 non-null int64
item         76071 non-null int64
avaliação    76071 non-null int64
dtypes: int64(3)
memory usage: 1.7 MB
```

1.8 Cálculo das métricas nos algoritmos

```
In [40]: from collections import defaultdict

         from surprise import Reader, Dataset
         from surprise import KNNBasic, KNNWithMeans, KNNWithZScore, KNNBaseline
         from surprise import SVD, SVDpp, NMF
         from surprise import NormalPredictor, BaselineOnly
         from surprise import CoClustering, SlopeOne
         from surprise.model_selection import cross_validate, KFold, GridSearchCV

In [6]: random.seed(0)
        np.random.seed(0)

        reader = Reader(line_format='user item rating', sep=',', rating_scale=(1,5))
        data = Dataset.load_from_file('tourismn.csv', reader=reader)

In [7]: def crossvalidate(algorithm):
            return cross_validate(algorithm, data, measures=['RMSE'], cv=5, verbose=False)

In [8]: def calculateRMSEFinalMean(algorithm):
            return np.mean(algorithm['test_rmse'])

In [9]: def precision_recall_at_k(predictions, k=10, threshold=3.5):
            # First map the predictions to each user.
            user_est_true = defaultdict(list)
            for uid, _, true_r, est, _ in predictions:
                user_est_true[uid].append((est, true_r))

            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
```

```
In [10]: kf = KFold(n_splits=5)

         # k = items
         def precs(algo):
             precisions_ = []
             for trainset, testset in kf.split(data):
                 algo.fit(trainset)
                 predictions = algo.test(testset)
                 precisions, recalls = precision_recall_at_k(predictions, k=5, threshold=4)

                 # Precision and recall can then be averaged over all users
                 precisions_.append(sum(prec for prec in precisions.values()) / len(precisions)

             return np.mean(precisions_)

         def recs(algo):
             recalls_ = []
             for trainset, testset in kf.split(data):
                 algo.fit(trainset)
                 predictions = algo.test(testset)
                 precisions, recalls = precision_recall_at_k(predictions, k=5, threshold=4)

                 # Precision and recall can then be averaged over all users
                 recalls_.append(sum(rec for rec in recalls.values()) / len(recalls))

             return np.mean(recalls_)
```

```
In [11]: sim_options_cosine = {'name': 'cosine','user_based': False}
         sim_options_msd = {'name': 'msd','user_based': False}
         sim_options_pearson = {'name': 'pearson','user_based': False}
         sim_options_baseline = {'name': 'pearson_baseline','user_based': False, 'shrinkage': 0
```

```
In [12]: knn_1 = KNNBasic(sim_options=sim_options_cosine, verbose=False)
```

```
kNNBasicCos = crossvalidate(knn_1)
rmse_kNNBasicCos = calculateRMSEFinalMean(kNNBasicCos)
pandr_kNNBasicCos = (precs(knn_1), recs(knn_1))
```

In [13]:
```
knn_2 = KNNBasic(sim_options=sim_options_msd, verbose=False)

kNNBasicMSD = crossvalidate(knn_2)
rmse_kNNBasicMSD = calculateRMSEFinalMean(kNNBasicMSD)
pandr_kNNBasicMSD = (precs(knn_2), recs(knn_2))
```

In [14]:
```
knn_3 = KNNBasic(sim_options=sim_options_pearson, verbose=False)

kNNBasicPearson = crossvalidate(knn_3)
rmse_kNNBasicPearson = calculateRMSEFinalMean(kNNBasicPearson)
pandr_kNNBasicPearson = (precs(knn_3), recs(knn_3))
```

In [15]:
```
knn_4 = KNNBasic(sim_options=sim_options_baseline, verbose=False)

kNNBasicPearsonB = crossvalidate(knn_4)
rmse_kNNBasicPearsonB = calculateRMSEFinalMean(kNNBasicPearsonB)
pandr_kNNBasicPearsonB = (precs(knn_4), recs(knn_4))
```

In [16]:
```
knn_5 = KNNWithMeans(sim_options=sim_options_cosine, verbose=False)

kNNMeansCos = crossvalidate(knn_5)
rmse_kNNMeansCos = calculateRMSEFinalMean(kNNMeansCos)
pandr_kNNMeansCos = (precs(knn_5), recs(knn_5))
```

In [17]:
```
knn_6 = KNNWithMeans(sim_options=sim_options_msd, verbose=False)

kNNMeansMSD = crossvalidate(knn_6)
rmse_kNNMeansMSD = calculateRMSEFinalMean(kNNMeansMSD)
pandr_kNNMeansMSD = (precs(knn_6), recs(knn_6))
```

In [18]:
```
knn_7 = KNNWithMeans(sim_options=sim_options_pearson, verbose=False)

kNNMeansPearson = crossvalidate(knn_7)
rmse_kNNMeansPearson = calculateRMSEFinalMean(kNNMeansPearson)
pandr_kNNMeansPearson = (precs(knn_7), recs(knn_7))
```

In [19]:
```
knn_8 = KNNWithMeans(sim_options=sim_options_baseline, verbose=False)

kNNMeansPearsonB = crossvalidate(knn_8)
rmse_kNNMeansPearsonB = calculateRMSEFinalMean(kNNMeansPearsonB)
pandr_kNNMeansPearsonB = (precs(knn_8), recs(knn_8))
```

In [20]:
```
knn_9 = KNNWithZScore(sim_options=sim_options_cosine, verbose=False)

kNNZCos = crossvalidate(knn_9)
rmse_kNNZCos = calculateRMSEFinalMean(kNNZCos)
pandr_kNNZCos = (precs(knn_9), recs(knn_9))
```

```
In [21]: knn_10 = KNNWithZScore(sim_options=sim_options_msd, verbose=False)

         kNNZMSD = crossvalidate(knn_10)
         rmse_kNNZMSD = calculateRMSEFinalMean(kNNZMSD)
         pandr_kNNZMSD = (precs(knn_10), recs(knn_10))

In [22]: knn_11 = KNNWithZScore(sim_options=sim_options_pearson, verbose=False)

         kNNZPearson = crossvalidate(knn_11)
         rmse_kNNZPearson = calculateRMSEFinalMean(kNNZPearson)
         pandr_kNNZPearson = (precs(knn_11), recs(knn_11))

In [23]: knn_12 = KNNWithZScore(sim_options=sim_options_baseline, verbose=False)

         kNNZPearsonB = crossvalidate(knn_12)
         rmse_kNNZPearsonB = calculateRMSEFinalMean(kNNZPearsonB)
         pandr_kNNZPearsonB = (precs(knn_12), recs(knn_12))

In [24]: knn_13 = KNNBaseline(sim_options=sim_options_cosine, verbose=False)

         kNNBaseCos = crossvalidate(knn_13)
         rmse_kNNBaseCos = calculateRMSEFinalMean(kNNBaseCos)
         pandr_kNNBaseCos = (precs(knn_13), recs(knn_13))

In [25]: knn_14 = KNNBaseline(sim_options=sim_options_msd, verbose=False)

         kNNBaseMSD = crossvalidate(knn_14)
         rmse_kNNBaseMSD = calculateRMSEFinalMean(kNNBaseMSD)
         pandr_kNNBaseMSD = (precs(knn_14), recs(knn_14))

In [26]: knn_15 = KNNBaseline(sim_options=sim_options_pearson, verbose=False)

         kNNBasePearson = crossvalidate(knn_15)
         rmse_kNNBasePearson = calculateRMSEFinalMean(kNNBasePearson)
         pandr_kNNBasePearson = (precs(knn_15), recs(knn_15))

In [27]: knn_16 = KNNBaseline(sim_options=sim_options_baseline, verbose=False)

         kNNBasePearsonB = crossvalidate(knn_16)
         rmse_kNNBasePearsonB = calculateRMSEFinalMean(kNNBasePearsonB)
         pandr_kNNBasePearsonB = (precs(knn_16), recs(knn_16))

In [28]: svd_ = SVD(verbose=False)

         svd = crossvalidate(svd_)
         rmse_svd = calculateRMSEFinalMean(svd)
         pandr_svd = (precs(svd_), recs(svd_))

In [29]: svdpp_ = SVDpp(verbose=False)
```

```
svdpp = crossvalidate(svdpp_)
rmse_svdpp = calculateRMSEFinalMean(svdpp)
pandr_svdpp = (precs(svdpp_), recs(svdpp_))
```

```
In [30]: nmf_ = NMF(verbose=False)

         nmf = crossvalidate(nmf_)
         rmse_nmf = calculateRMSEFinalMean(nmf)
         pandr_nmf = (precs(nmf_), recs(nmf_))
```

```
In [31]: baseline_ = BaselineOnly(verbose=False)

         baselineOnly = crossvalidate(baseline_)
         rmse_baselineOnly = calculateRMSEFinalMean(baselineOnly)
         pandr_baselineOnly = (precs(baseline_), recs(baseline_))
```

```
In [32]: cocluster_ = CoClustering(verbose=False)

         coClustering = crossvalidate(cocluster_)
         rmse_coClustering = calculateRMSEFinalMean(coClustering)
         pandr_coClustering = (precs(cocluster_), recs(cocluster_))
```

```
In [33]: slope_ = SlopeOne()

         slopeOne = crossvalidate(slope_)
         rmse_slopeOne = calculateRMSEFinalMean(slopeOne)
         pandr_slopeOne = (precs(slope_), recs(slope_))
```

```
In [35]: dict_ = {
             'Algorithm': names,
             'RMSE': scoresRMSE,
             'Precision': scoresPrecision,
             'Recall': scoresRecalls
         }
```

```
In [36]: df = pd.DataFrame(data=dict_)
```

1.8.1 Resultados

Relembrando as métricas para avaliação, o algoritmo mais apropriado para o caso em questão é aquele que possui **o resultado do RMSE mais baixo** e **os cálculos da *precision* e do *recall*, ambos, mais altos**. É importante fixar que temos como objetivo encontrar um algoritmo que esteja nivelado nessas duas exigências, não adianta por exemplo ter um RMSE extramamente baixo e um *recall* extramamente baixo também.

```
In [37]: df.sort_values(by=['RMSE']).head()
```

Out[37]:

	Algorithm	RMSE	Precision	Recall
18	Baseline Only	0.887368	0.856771	0.919997
16	SVD	0.904947	0.863477	0.871282
17	SVDpp	0.909414	0.864176	0.889405
15	kNN Baseline - Pearson B	0.984349	0.860136	0.881130
7	kNN Means - Pearson B	0.985400	0.849444	0.907843

In [38]:
```
df.sort_values(by=['Precision'], ascending=False)
```

Out[38]:

	Algorithm	RMSE	Precision	Recall
21	NMF	1.132062	0.902238	0.647081
13	kNN Baseline - MSD	1.021009	0.878516	0.806803
12	kNN Baseline - Cosine	1.022989	0.877679	0.807222
4	kNN Means - Cosine	1.026102	0.872733	0.820434
19	CoClustering	1.052364	0.872687	0.814877
9	kNN Z - MSD	1.029817	0.872260	0.823811
20	SlopeOne	1.031187	0.872003	0.818548
14	kNN Baseline - Pearson	1.044356	0.871899	0.823381
8	kNN Z - Cosine	1.032567	0.871584	0.825772
5	kNN Means - MSD	1.026867	0.871545	0.820742
6	kNN Means - Pearson	1.050895	0.865610	0.837697
17	SVDpp	0.909414	0.864176	0.889405
10	kNN Z - Pearson	1.047561	0.864059	0.839341
16	SVD	0.904947	0.863477	0.871282
15	kNN Baseline - Pearson B	0.984349	0.860136	0.881130
0	kNN Basic - Cosine	1.067450	0.857239	0.863415
18	Baseline Only	0.887368	0.856771	0.919997
1	kNN Basic - MSD	1.068189	0.856199	0.864739
11	kNN Z - Pearson B	0.986441	0.849968	0.910767
7	kNN Means - Pearson B	0.985400	0.849444	0.907843
2	kNN Basic - Pearson	1.095482	0.846264	0.887821
3	kNN Basic - Pearson B	1.026674	0.835822	0.951032

Analisando as tabelas acima, podemos observar que os algoritmos **SVD** e **SVDpp** possuem um bom índice nas duas exigências, sendo, assim, os escolhidos.