University of California, Davis

STA 160 Project

Rui Li / 998841986
Shunguan Mai / 999073944
Shengjie Shi / 999078428

Supervised by
Professor Duncan Temple-Lang

Abstract

An accurate item recommendation system is crucially important to ecommerce companies like Amazon, Netflix and eBay, etc. Its of merchants greatest interest to showcase more relevant items to increase customers willingness to purchase and sales revenue.

This project will mainly focus on developing movie recommendation system by using two popular strategies: collaborative filtering (CF) and content-based filtering. We will apply Alternating Least Square (ALS) method in collaborative filtering, which is an algorithm that implicitly incorporated with latent factors on matrix factorization models; and we will also suggest movies based on movie overviews and movie posters in content-based filtering. Our goal is to study the performance of different recommendation strategies. The dataset we used in this project is obtained from MovieLens research site. The dataset contains 100004 ratings and 1296 tag applications across 9125 movies created by 671 unique users. One drawback of this dataset is it does not include any information about overviews and posters of movies, therefore we have used the Movie Database API to query more necessary information.

Collaborative Filtering for Movie Recommendation

In general, movie recommendation systems are based on one of two strategies: *content filtering* approach and *collaborative filtering* approach. The content filtering approach creates a profile for each user or product to characterize its nature. Obviously, this approach requires gathering external information that might not be available or not easy to obtain. Collaborative filtering analyzes relationships between users and interdependencies among products to identify new user-item associations.

Two primary areas of collaborative filtering are *neighborhood methods* and *latent factor models*. Neighborhood methods are centered on computing the relationships between items or, alternatively, between users. Latent factor models are an alternative approach that tries to explain the ratings by characterizing both item (in this context, movie) and user factors inferred from the ratings patterns. For movies, the discovered factors might measure obvious dimensions such as comedy versus drama, amount of action, or orientation to children; less well-defined dimensions such as depth of character development or quirkiness; or completely uninterpretable dimensions. For users, each factor measures how much the user likes movies that score high on the corresponding movie factor.

Some of the most successful realizations of latent factor models are based on matrix factorization. In its basic form, matrix factorization characterizes both items and users by vectors of factors inferred from item rating patterns.

Problem Formulation

Given n_m movies and n_u users in the dataset, denote each movie as m and each user as u. Matrix factorization models map both users and movies to a joint latent factor space of dimensionality f, such that user-movie interactions (i.e. ratings) are modeled as inner products in that space. Accordingly, each movie m is associated with a vector (i.e. latent factor for this movie) $q_m \in \mathbb{R}^f$, and each user u is associated with a vector $p_u \in \mathbb{R}^f$. For a given movie m, elements of q_m are real numbers and measure the extent to which the movie possesses these factors. For a given user u, elements of p_u are real numbers and measure the extent of interest the user has in movies that are high on the corresponding factors. This setup characterizes a rating of a movie, r_{um}, given by a user as the inner product of q_m and p_u, i.e.

$$r_{um} = q_m^T \cdot p_u \tag{1}$$

The goal here is to learn the latent factors q_m and p_u. We can formulate the problem as

$$\min_{q_m, p_u \in \mathbb{R}^f} F(q_m, p_u) := \sum_{(u,m)\in\kappa} (r_{um} - q_m^T\ p_u)^2 + \lambda(||q_m||^2 + ||p_u||^2) \tag{2}$$

Here, the κ is set of the (u, m) pairs for which r_{um} is known. That is, κ contains all pairs of (u, m) if user u rated movie m. The system learns the model by fitting the previously observed ratings. However, the goal is to generalize those previous ratings in a way that predicts future, unknown ratings. Thus, the system should avoid overfitting the observed data by regularizing the learned parameters, whose magnitudes are penalized. The constant λ controls the extent of regularization and is usually determined by cross-validation.

There are two approaches to minimize Equation 2: *stochastic gradient descent* and *alternating least squares* (ALS). Because both q_m and p_u are unknowns, Equation 2 is not convex. However, if we fix one of the unknowns, the optimization problem becomes quadratic and can be solved optimally. Thus, ALS techniques rotate between fixing the q_m's and fixing the p_u's. When all p_u's are fixed, the system recomputes the q_m's by solving a least-squares problem, and vice versa. This ensures that each step decreases Equation 2 until convergence.

Algorithm 1 *Collaborative Filtering*

1: Initialize q_m and p_u in $\mathbb{R}^f$
2: Minimize $F(q_m, p_u)$ using stochastic gradient descent
$e_{um} := r_{um} - q_m^T\, p_u$
$q_m \leftarrow q_m - \gamma \cdot (e_{um} \cdot p_u - \lambda \cdot q_m)$
$p_u \leftarrow p_u - \gamma \cdot (e_{um} \cdot q_m - \lambda \cdot p_u)$
3: Predict a user's rating by $r_{um} = q_m^T \cdot p_u$

The *collaborative filtering algorithm* is stated as follows:
The outcome of this algorithm is a dense matrix where each entry is either a predicted rating by collaborative filtering algorithm, or a rating given by the user. The latent factor matrix for users, P, and latent factor matrix for movies, Q, are

$$P = \begin{bmatrix} — & (p^{(1)})^T & — \\ — & (p^{(2)})^T & — \\ & \vdots & \\ — & (p^{(n_u)})^T & — \end{bmatrix},\ Q = \begin{bmatrix} — & (q^{(1)})^T & — \\ — & (q^{(2)})^T & — \\ & \vdots & \\ — & (q^{(n_m)})^T & — \end{bmatrix}$$

So the matrix of ratings is

$$\widetilde{Y} = PQ^T = \begin{pmatrix} (p^{(1)})^T(q^{(1)}) & (p^{(1)})^T(q^{(2)}) & \cdots & (p^{(1)})^T(q^{(n_m)}) \\ (p^{(2)})^T(q^{(1)}) & (p^{(2)})^T(q^{(2)}) & \cdots & (p^{(2)})^T(q^{(n_m)}) \\ \vdots & \vdots & \ddots & \vdots \\ (p^{(n_u)})^T(q^{(1)}) & (p^{(n_u)})^T(q^{(2)}) & \cdots & (p^{(n_u)})^T(q^{(n_m)}) \end{pmatrix}$$

We explore the *Apache Spark* and use it to perform the prediction, in which the function `pyspark.mllib.recommendation.`**`ALS`** is invoked. This function allows us to tune certain parameters, and two of them should be noted: `rank` and `lambda_`. `rank` indicates the number of latent factors, and `lambda_` is the magnitude of regularization parameter. After comparing differnt choices of `rank` and `lambda_`, we discover that `rank` $= 8$ and `lambda_` $= 0.25$ yields the lowest test $RMSE = 0.88$. To complete the recommendation, we sort the ratings for each user (i.e. each row of $\widetilde{Y}$) in descending order and make recommendation of, say, top 20 movies of highest predicted ratings to this user.

Content-based Filtering for Movie Recommendation

Besides the collaborative filtering for movie recommendation, content-based filtering is another method of involving users watching histories to make recommendations. Compared with collaborative filtering using the historical ratings to calculate the similarity

between each user, content-based filtering algorithm will involve more information that is related to the content of each movie in users viewing profile, such as movie overview or users review. This method tries to figure out the more favorable preference for each user.

Algorithm Implementation

This algorithm relies on the two basic models, which are word2vec and support vector machine(SVM). The word2vec model was developed by Mikolov et al. 2013, fully named as Vector Representations of Words. Generally, Word2Vec (W2V) is an algorithm that figures out how to place words on a chart in such a way that their location is determined by their meaning. This means that words with similar meanings will be clustered together. The really famous example is [king] - [man] + [woman] = [queen] (another way of thinking about this is that [king] - [queen] is encoding just the gendered part of [monarch]). Equivalently, by implementing this algorithm on the overview of movie, each user will have their own movie dictionary based on the movie overview they have been watched before. To improve the efficiency of clustering, Googles pre-trained model will also combine with word2vec model. It includes word vectors for a vocabulary of 3 million words and phrases that they trained on roughly 100 billion words from a Google News dataset. The vector length is 300 features.

Another wide-used approach is support vector machine(SVM). Support Vector Machine (SVM) is a supervised machine learning algorithm that can be more commonly employed in classification problems. Here, the essential part is training a SVM for each movie and nd a separating hyperplane between the users who liked the movie and users who didnt based on the rating score. Then we use the linear kernel function to map the original data into the high dimension hyperplane. The algorithm described above is the classification format, but here we use the regression format. The matrix from word2vec of overviews is the X, and the vector of rating score is the y.

Process Clarification

Data gathering

The first five rows of the original data from themoviedb.org shown below. This dataset contains 9066 movies, and the imdbId can be used to obtain the movie overview by using json package in Python.

	userId	movieId	rating	timestamp	title	genres	imdbId	tmdbId
0	1	30	2.5	1260759144	Dangerous Minds (1995)	Drama	112792	9909.0
1	1	1128	2.0	1260759185	Escape from New York (1981)	Action|Adventure|Sci-Fi|Thriller	82340	1103.0
2	1	1171	4.0	1260759205	Cinema Paradiso (Nuovo cinema Paradiso) (1989)	Drama	95765	11216.0
3	1	1342	2.0	1260759131	Cape Fear (1991)	Thriller	101540	1598.0
4	1	1262	2.0	1260759151	Deer Hunter, The (1978)	Drama|War	77416	11778.0

Figure 1: The preview of original dataset

Preprocess

After got the overview for each movie, the overview needs to be preprocessed before entering the model fitting process.

If each overview is treated as one long string, the major steps of preprocess are as following:

1. Removing the stop words for each paragraph of overview
2. Change all the uppercase letters to lowercase letters
3. Replace all the punctuation mark by space
4. Construct our process as a two-step pipeline: First is to vectorize the overview string by using word2vec model, and next is to train the SVR model using the matrix from word2vec model as the X, and the rating score as the y
5. Sample 100 movies in the dataset to be new movies, using these movies to give users recommendation, and use the model above for each user to predict the rating scores for those 100 movies
6. Sort the top ten recommendations to be the result

Result Display

In order to view more vivid result, the poster is used to represent the movie. For example, here are top 8 rated movies from user 1s watching profile and the corresponding top 8 recommendations. From the wordcloud pictures(left is historically watched content text, right is the recommended content) that there are some meaning related word, such as prisonerpolice, computerprogram—remote.

Figure 2: The Wordcloud of Overview from User's Profile

Figure 3: The Wordcloud of Overview from Recommendation

Movie Recommendation Based on Poster Images with Neural Network

Movie poster often time suggests the theme or the content of a movie. For example, posters of animation movies usually portray cartoon characters with vivid colors; posters of horror movies always have a close-up of a dreadful face with dark background color. Intuitively, we can also recommend movies based on movie posters.

To train and to recognize the movie posters, we used the *VGG16* image classifier which is developed by the Visual Geometry at the University of Oxford. As the name suggests, VGG16 consists of 16 layers. Several variation exists. It repeats the pattern of 2 convolution layers followed by 1 dropout layers until the fully connected layer at the end.

The followings are the procedure to build the recommendation system based on movie posters.

1. Query movie posters
2. Extract the features from posters
3. Recommendation based similarities of poster features

Load and Preprocess movie posters

There are 9,060 unique movies in the training dataset and each movie is linked to a IMDB ID number. We used the IMDB ID's to query the movie posters through The Movie Database (TMDB) API and retrieve them to local directory. We substituted a

grey image for a movie poster for every failure of retrieving the poster from the API. The reason to use a grey image is that its RGB value is (128, 128, 128), it is the median of color values, which is robust against future feature similarity calculation.

Extract Features from Posters

Later, we resized the image to 224 × 224 and normalized the color channel, and eventually feed the preprocess image to VGG16 model from Keras neural network library. Each movie posters will be converted to a vector with 25,088 features.

Calculate Similarities

After extracting the features from all unique movies from the training dataset, we will have a 9,060 × 25,088 matrix where each row is the features vector for a specific movie. Finally, for a new movie, we can preprocess the new movie poster with the same steps above, then calculate its cosine similarity with each row from our matrix and recommend the corresponding most similar movie.

Figure 4: Input Movie Posters

Figure 5: Recommended Movie Posters

Advantages

1. This recommendation is based on minimal amount of information provided by movies and users. Without the presence of a users movie-watched history and a movies content, casts, budgets, the model can recommend movies simply by finding similar movie posters. Therefore, this algorithm is less expensive because of less data usage. The recommendation is less expensive considering it requires less user data.

2. The recommendation outputs share visual similarities on their posters, it can raise a users interest to click on the recommended items.

Disadvantage

1. This algorithm is crucially based on the assumption that movie posters can be a good representative of movies, such as they can reveal the style, the genre, and the content of movies. However, two completely different movies would also have a chance to share similar posters, such situation will violate the principle assumption and caused inaccuracy in recommendation.

2. This algorithm ignores many other user preference information. For example, a fan of Leonardo DiCaprio may not always find this recommendation engine useful, because the algorithm always suggests movies with similar posters, not movies performed by the same actor.

Conclusion

We have experienced two major recommendation strategies through this project: the collaborative filtering and the content-based filtering. In collaborative filtering, we used the Alternating Least Square algorithm; in content-based filtering, we used the Word2Vec plus SVM to suggest movies based on movie overviews, and deep neural network to recommend movies based on movie poster similarities. Today, collaborative filtering is the most popular method because it is more scalable and it considers the preference of each user to produce a more accurate prediction. However, collaborative filtering has a poor performance if there is little data about users ratings; in such situation, content-based filtering can be used as an alternative to suggest movies since the algorithm only requires the overviews or the posters of the movies. In summary, each algorithm has its own advantages and disadvantages, and we should determine the proper scenarios to apply the algorithms.

Appendix

Reflection

The collaborative filtering model has one noticeable drawback: cold-start problem. To successfully run this collaborative filtering algorithm, one needs to possess huge amount of user ratings data. However, for a newly kicked off project, it is nearly impossible to collect such data. For the purpose of this project, we can use existing dataset to accomplish this, but in reality companies need to tackle this problem.

The movie overview is the summary of main story with relative background introduction. It includes most of characteristics that will show in the movie and involves more information that can objectively represent the movie itself than other information in the original dataset. Sometimes, even user has watched the movies and rated them by chance, they will not leave any review to show their preference. However, by analyzing the overview of each film in user's profile, the entertainment related company such as Netflix or Amazon can still to make recommendation to those kind of users, which remedy the unicity by using rating score.

On the other hand, it's hard to test the recommendation accuracy of this algorithm because of limited time. So in the future study, trying to find a reasonable test method will be the next thing should carry out.

In real world, recommendation system is more complicated than what we have done so far. For example, Netflixs recommendation system is a combination of different machine learning techniques. Two of the most notable techniques are matrix factorization and the so-called temporal dynamics to perform collaborative filtering. We also sense the potential to combine the three algorithms in this project. For example, we can select movies with similar overviews after collaborative filtering.

Code Snippets

```
# Content-based appraoch: Word2Vec and SVM

class MeanEmbeddingVectorizer(object):
def __init__(self, word2vec):
self.word2vec = word2vec
```

```
6  self.dim = len(word2vec.values())
7
8  def fit(self, X, y):
9  return self
10
11 def transform(self, X):
12 return np.array([np.mean([self.word2vec[w] for w in words if
    ↪ w in self.word2vec] or [np.zeros(self.dim)], axis=0) for
    ↪ words in X])
13
14
15 def split_word(x):
16 if pd.notnull(x):
17 return nltk.word_tokenize(x)
18 else:
19 return "None"
20
21 # read the file
22 user_table = pd.read_csv("df.csv", encoding='ISO-8859-1')
23 imdb_table = pd.read_csv("9066.csv")
24 # sample 100 random movies to do the prediction
25 imdb_table_sample =
    ↪ imdb_table.sample(n=100).reset_index(drop=True)
26 # load w2v model
27 model =
    ↪ gensim.models.KeyedVectors.load_word2vec_format('GoogleNews-vectors-
    ↪ binary=True)
28 w2v = {w: vec for w, vec in zip(model.index2word,
    ↪ model.syn0)}
29 # pipeline
30 pipeline = Pipeline([("word2vec vectorizer",
    ↪ MeanEmbeddingVectorizer(w2v)), ("linear svr",
    ↪ SVR(kernel="linear"))])
31
32 user_table_splitted = user_table.groupby("userId")
33 user_all = list(user_table_splitted.grouper)
```

```
result_dict = []

for user in user_all:
temp_table = user_table_splitted.get_group(user)
merged_table = pd.merge(temp_table, imdb_table)
merged_table.insert(merged_table.shape[1],
 ↪ "overview_splitted", "")
merged_table['overview_splitted'] =
 ↪ merged_table['overview'].apply(split_word)
model = pipeline.fit(merged_table['overview_splitted'],
 ↪ merged_table['rating'])
predicted =
 ↪ model.predict(imdb_table_sample['overview'].apply(split_word))
index = [imdb_table_sample.iloc[i[0]]['imdbId'] for i in
 ↪ sorted(enumerate(predicted), key=lambda x: x[1],
 ↪ reverse=True)][:10]
result_dict.append({'userId': user, 'best_imdbId': index})

pd.DataFrame.from_dict(result_dict).to_csv("bestImdbId.csv",
 ↪ index=False, encoding='utf-8')
```

```
# Collaborative filtering

sc = pyspark.SparkContext()

full_data = sc.textFile(path.join(getcwd(), "ml-latest",
 ↪ "ratings.csv"))
small_data_header = full_data.take(1)[0]

# Parse the data and read in one line of form (user, movie,
 ↪ rating) each time.
small_ratings_data = (full_data.filter(lambda line: line !=
 ↪ small_data_header)
.map(lambda line: line.split(","))
.map(lambda tokens: (tokens[0], tokens[1],
 ↪ tokens[2])).cache())
```

```
12
13 complete_movies_raw_data = sc.textFile(path.join(getcwd(),
    ↪ "ml-latest", "movies.csv"))
14 complete_movies_raw_data_header =
    ↪ complete_movies_raw_data.take(1)[0]
15
16 # Parse
17 complete_movies_data =
    ↪ (complete_movies_raw_data.filter(lambda line: line !=
    ↪ complete_movies_raw_data_header)
18 .map(lambda line: line.split(","))
19 .map(lambda tokens:
    ↪ (int(tokens[0]),tokens[1],tokens[2])).cache())
20
21 complete_movies_titles = complete_movies_data.map(lambda x:
    ↪ (int(x[0]), x[1]))
22
23 movie_ID_with_ratings_RDD = small_ratings_data.map(lambda x:
    ↪ (x[1], x[2])).groupByKey().mapValues(list)
24 movie_ID_with_avg_ratings_RDD =
    ↪ movie_ID_with_ratings_RDD.map(get_counts_and_averages)
25 movie_rating_counts_RDD =
    ↪ movie_ID_with_avg_ratings_RDD.map(lambda x: (int(x[0]),
    ↪ x[1][0]))
26
27 # Obtain all movie id's and a list of ratings
28 movie_id = movies.movieId
29 rating = range(1, 6)
30
31 new_user_ID = options.u
32 num_movie = options.num_movie
33 num_rating = num_movie
34
35 new_user_ratings = zip(list(np.repeat(new_user_ID,
    ↪ num_movie)),
```

```
36 list(np.random.choice(movie_id, num_movie)),
    ↪ list(np.random.choice(rating, num_rating)))
37
38 new_user_ratings_RDD = sc.parallelize(new_user_ratings)
39 complete_data_with_new_ratings_RDD =
    ↪ small_ratings_data.union(new_user_ratings_RDD)
40
41 # Everytime a new user is added to the dataset, the model
    ↪ needs to be trained again.
42 new_ratings_model =
    ↪ ALS.train(complete_data_with_new_ratings_RDD, best_rank,
43 seed = seed, iterations = iterations, lambda_ =
    ↪ regularization_parameter)
44
45 new_user_ratings_ids = map(lambda x: x[1], new_user_ratings)
46
47 new_user_unrated_movies_RDD =
    ↪ complete_movies_data.filter(lambda x: x[0] not in
    ↪ new_user_ratings_ids).map(lambda x: (new_user_ID, x[0]))
48
49 # Use the input RDD, new_user_unrated_movies_RDD, with
    ↪ new_ratings_model.predictAll()
50 # to predict new ratings for the movies that new user has
    ↪ not seen.
51 new_user_recommendations_RDD =
    ↪ new_ratings_model.predictAll(new_user_unrated_movies_RDD)
52
53 new_user_recommendations_rating_RDD =
    ↪ new_user_recommendations_RDD.map(lambda x: (x.product,
    ↪ x.rating))
54 new_user_recommendations_rating_title_and_count_RDD =
    ↪ (new_user_recommendations_rating_RDD.join(complete_movies_titles)
55 .join(movie_rating_counts_RDD))
56
57 new_user_recommendations_rating_title_and_count_RDD =
    ↪ (new_user_recommendations_rating_title_and_count_RDD
```

```
.map(lambda r: (r[1][0][1], r[1][0][0], r[1][1])))

top_movies =
 ↪ new_user_recommendations_rating_title_and_count_RDD.filter(lambda
 ↪ r: r[2] >= 25).takeOrdered(25, key = lambda x: -x[1])
```

```
# Convolutional Neural Network

#preprocess image
#Courtesy to:
 ↪ http://www.datasciencecentral.com/profiles/blogs/deep-learning-meets
img = [0]*len(df_image)
x = [0]*len(df_image)
for i in range(len(df_image)):
img[i] = kimage.load_img(poster_path + str(i) + ".jpg",
 ↪ target_size=(224, 224))
x[i] = kimage.img_to_array(img[i])
x[i] = np.expand_dims(x[i], axis=0)
x[i] = preprocess_input(x[i])

# image_top=False removes final connected layers
model = VGG16(include_top=False, weights='imagenet')
http://www.datasciencecentral.com/profiles/blogs/deep-learning-meets-rec
# create prediction
pred = [0]*len(df_image)
pred_norm = [0]*len(df_image)
matrix_res = np.zeros([len(df_image), 25088])
for i in range(len(df_image)):
pred[i] = model.predict(x[i]).ravel()
matrix_res[i,:] = pred[i]
```

Bibliography

[1] M. (2016, October 18). MovieLens: recommended for education and development. Retrieved June 12, 2017, from https://grouplens.org/datasets/movielens/

[2] Rosenthal, E. (2016, December 5). Using Keras' Pretrained Neural Networks for Visual Similarity Recommendations. Retrieved June 12, 2017, from http://blog.ethanrosenthal.com/2016-/12/05/recasketch-keras/

[3] Ma, W. (2017, April 21). Deep Learning Meets Recommendation Systems. Retrieved June 12, 2017, from http://www.datasciencecentral.com/profiles/blogs/deep-learning-meets-recommendation-systems

[4] Dianes, J. A. (2015, July 07). Building a Movie Recommendation Service with Apache Spark & Flask - Part 1. Retrieved from https://www.codementor.io/jadianes/building-a-recommender-with-apache-spark-python-example-app-part1-du1083qbw