Learning about Media Users from Movie Rating Data

by

Lantian Chen

B.S. Computer Science and Engineering,

MIT 2019

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

AT THE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

SEPTEMBER 2020

Author:_____

Department of Electrical Engineering and Computer Science

August 24, 2020

Certified by:_____

Phillip Isola

Assistant Professor at MIT EECS, MIT Thesis Supervisor

Certified by:_____

Jian Li

Principal Data Scientist at Sky UK, 6A Thesis Supervisor

Approved by:_____

Katrina LaCurts

Chair, Master of Engineering Thesis Committee

Learning about Media Users from Movie Rating Data

by

Lantian Chen

Submitted to the Department of Electrical Engineering and Computer Science
On August 24, 2020 in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

In this thesis, we study techniques of machine learning for media users who submitted movie ratings to the MovieLens dataset --- a project inspired by Sky UK's own business problems I encountered during my internship there. It follows the "feature engineering" paradigm, compared to the "deep learning" paradigm, through three stages: Feature Engineering, Clustering and Recommendation, each being a classic machine learning problem. For each step, I am introducing the common, relevant methods, along with my own designed models on top of available tools and experiments on the MovieLens data on the Google Cloud Platform. Due to the open-ended nature of all three problems, we don't have quantifiable conclusions on which methods would prove the best; instead, presented here is some learning on the trade-offs and suitability for these designs.

Thesis Supervisor: Phillip Isola

Title: Bonnie & Marty (1964) Tenenbaum CD Assistant Professor

Acknowledgements

This thing would not have been possible if not for the following people, to whom I'm expressing my gratitude:

Jian Li, for being supportive, patient and trusting throughout my entire internship and, of course, helping me find a future career at Sky;

Phillip Isola, for being the best supervisor an M.Eng. student could ask for and offering knowledge, help and encouragement every time (I know I didn't exceed his expectation, but he certainly did mine);

Joe Boadi, for convincing me I could survive in London, at Sky and on Google Cloud Platform;

Myriam, Katrina, Vera, Brandi, Kathy, Prof. Palacios, for their support from EECS during this unusual time;

My cousin Jin Qiao, her son/my nephew Adam, for their relentless cheerleading and unwavering cuteness;

South London Stags RFC, John Burgess, for being two good things happening at the end and lifting my mood unexpectedly;

Jay, Vivek, Lucas, for entering my life a bit earlier and, too, making it a bit better;

Shariann, AL, Amanda, An, Natalie and Yidan, for waiting for me at the finishing line.

And thanks to these places on earth:

Chiswick House, Duke's Meadow and Waft Coffee,

Along with the cover band **Boyce Avenue**.

For their sake, I managed to not have lost my mind through a global pandemic and, apparently, an MIT thesis.

I'm dedicating this work to **my grandfather**. Hope you've been in peace since February.

Contents

Chapter 1

Introduction

1.1 Sky's Problem: Promo Optimization

This thesis is generated as a result of my data science internship at Sky from 2020 February to 2020 August. Sky is the UK's largest pay-TV broadcaster with 12.5 million customers, as of 2018. **[1]** For a Sky viewer, a few minutes in an hour of TV programming will be scheduled for promotions, or promos — they are the advertisements for Sky's own content materials, such as movies or sport events. These promos have the potential to elevate the user's engagement for programs. However, without knowing the audience and targeting the right viewers, these promos may be ineffective when being shown to the uninterested people.

In the meantime, Sky has garnered two types of valuable data about its viewers:

- The detailed records of viewing activities (who watched what at what time) combined with descriptive tags on the watched programs (such as genre);
- A detailed model of the viewer profiles. Each user is mapped into a modeled profile, and each profile has more than 200 features (pieces of information about the user, such as gender, age, and income level).

Then naturally we can ask this question: **how can the available user data help us design better (even individualized) promo schedules — at best, automatically — and thus improve the user response rates and achieve the end goal of higher profits**? And this became the origin of my thesis work.

Why would this be important to Sky?

- **Financial impact**. A small fluctuation in user response rates can result in a difference of millions of pounds in revenue. All the investment needed is a machine learning pipeline and one or two engineers to monitor and update the results.
- **Operational model: data intelligence**. The more fundamental impact is for Sky to modify its operational model — from a manual-surveying, discretion-based process of decision making, to be an automatic, systematic and data-driven one. This promo problem is an example where data can be directly leveraged for business decisions, with a high certainty of positive impact. And in the future, this can become a more reliable and efficient model — a norm — for data scientists to integrated as an essential part of the business process, given the availability of user data (big data) and the rising demand for more tailored user experiences.

1.2 My Research Problem and the Thesis

Initially the research was conducted on Sky's dataset of user activities, but due to data confidentiality concerns, later I pivoted to focus my research on the classic movie rating dataset, MovieLens. Two datasets share much of similarity — they both record users' consumption of media content — but the difference is MovieLens data also contains rating info, whereas Sky's data contains the timing of each viewing record. (MovieLens has "timestamp" columns, but that is the timing when the user submitted the rating, rather than when watching the movie.) **Due to data confidentiality concerns, this thesis is based on only the MovieLens dataset.**

But on both datasets, my research problem boils down to: **what can we learn about the media users given the data of their media consumption and ratings?**

This is a classic data mining question — with open-ended goals. Below are a few possibilities:

- **Clustering.** How can we divide the users into subgroups, each group consisting of "similar" users by certain measure? And what are the salient features to describe the users in each subgroup?
- **Recommendation.** Given a particular user (or a group of users), what other unviewed movies (or TV programs) would be interesting? What should we recommend, not just the directly interesting materials, but also the programs the users don't expect they would be interested in?
- **Optimization.** For the "promo optimization" project, knowledge about the users is only an intermediate step, the end goal is to maximize revenue. So taking into account factors like the "revenue potential" of the programs or the restrictions of promo schedule slots, how would the knowledge of users result in an optimal, individualized promo schedule?

There are two approaches to solve this problem. The first one is a "manual" approach — **the feature engineering paradigm**. Basically, it breaks the whole machine learning pipeline into several stages and intermediate results. For our problem, we can break them into these three stages:

- **Feature Engineering.** (Chapter Two.) Transforming the raw data into a set of features for each individual user.
- **Clustering.** (Chapter Three.) Dividing the whole user set into cluster groups, based on individual feature vectors.
- **Recommendation.** (Chapter Four.) Making movie recommendations based on the clustering and individual information.

The other approach is **the deep learning paradigm**: we do only minimal data processing before feeding the data into the neural network structure and training the model to produce the end results. In our case, the end result is (or can be) movie recommendations. The trade-off between the two approaches is: the first feature engineering approach provides much more **transparency** and **explainability**, while the deep learning approach is more **automatic**, leverages much of the **complexity** within the neural network architecture's capacity, and can discover data patterns not discoverable or interpretable by humans. This will be explored briefly in the concluding Chapter Five.

1.3 Technical Complexities and Practical Concerns

During the first part of my research on Sky's dataset, some technical challenges emerged. (And these challenges were still relevant when I later turned to focus on the MovieLens lens.) As it turned out, the research work was not just about solving those research questions about the media users, it also contains many engineering complexities.

Performance Engineering & Computation Efficiency. The amount of computation required is not trivial. For a single day, on average there are more than 70 million viewing activity records (rows) for Sky's UK viewership. Sky is using Google Cloud Platform (GCP) as the computation power, but still I was required, as an engineer, to design efficient code and computing approaches. One example is using SQL directly, substituting computations processed by Pandas and Python, to increase speed. There are techniques online for machine learning-oriented data processing techniques. And there are GCP techniques to facilitate this. **[2]**

Building Efficient & Robust Systems. Another engineering concern was relevant in the project — to build a system as efficient and robust as possible. How do we design databases when a large amount of intermediate data is also generated? Can we design the system to be distributive, so that parallel computing can be achieved? This was more of a concern for Sky's dataset because of its sheer size. However, the MovieLens dataset can comfortably resides on the GCP.

Deployment of Results. This is another piece of complexity about Sky's dataset. I partnered with and report to another department, Content Supply Chain, which handles the business side of the problem. It is responsible for procuring and managing many sources of user data at Sky, and it collaborates with the data science group to leverage data mining and ultimately roll out the promo schedules for Sky's users. My research also needs to take into account how to participate in the deployment process — making it effective and cost-efficient.

Aside from these engineering complexities to make the system work, some other practical concerns also emerged to make the system more aligned with business needs:

Capturing Changes in User Behaviors. What the algorithm is trying to predict and influence is human behaviors, and they may not be stable. How sensitive is our algorithm's ability to detect a shift in users' preferences? Or for example, if a user's girlfriend moves into the house, can our system react to that quickly? Or if the user simply is just tired of repetitions and looking for something new? One immediate solution is to assign time-decaying weights, when compiling the user's historical profile, to emphasize recency. And how should we set the decaying rate? Or should we design some other more sophisticated mechanisms to detect changes and discern noises? Other related issues are seasonality (the user's preferences may be recurring and tied to some seasonal schedules, such as TV seasons and sport schedules) and trends (for example, the coronavirus pandemic is leaving a huge cultural imprint). Can the algorithm learn these "macro" factors? And possibly provide some "domain adaptation" when the broader TV culture moves? These are questions to be explored further for Sky's dataset after this thesis.

Parameter-Setting by Human vs. Machine. The model I am designing is not a cookie-cutter one, but one with many parameters involved responding to business or engineering needs. A recurring theme is: how should they be determined, manually or by machine learning? An intuition is that it's helpful to have a framework first with some pre-set parameters. And later in the experiment these parameters can be

further developed through machine learning or some scientific-approach. The trade-off is that human-set parameters are more immediately meaningful, or explainable, and can directly reflect discretionary business strategies; the machine-learned parameters can be more aligned with data and but always have the risk of overfitting. This plays out in every stage of the "feature engineering" paradigm, discussed in Chapters 2, 3 and 4.

And now, we're ready to dive deep into the dataset this thesis is based on, the MovieLens dataset, and start from the first stage of the "manual approach", feature engineering.

Chapter 2

MovieLens Data and Feature Engineering

2.1 The MovieLens Dataset

The MovieLens dataset was created in 1997 by GroupLens Research, a research lab in the Department of Computer Science and Engineering at the University of Minnesota. On MovieLens, real-world people can rate movies they've watched on a 5-star scale and use tags (generated by themselves) to describe those movies. The version of the dataset I'm using was generated on November 21, 2019, containing data provided by 162,541 users ever since January 09, 1995. **[3]** The dataset, with multiple versions released since its creation, was widely used by research and industry — for example, in 2014, it had more than 140,000 downloads, and more than 7,500 references to "movielens" appeared in Google Scholar. **[4]** MovieLens is thus one of the most classic data sources to study user movie preferences and machine-generated movie recommendations, which is the primary goal of this research work.

The dataset can be divided into two parts. The first part, which I'm referring to as "the base data", contains genre labels (coarse categorizing information) for all the available movies and all the movie ratings gathered from participating users. The other part, "the tagging data", includes more than 1K tag labels (fine categorizing information) and the associations between these tags and the movies.

Below are the specifics (the numbers in square brackets represent the numbers of data entries).

The base data:

- Users [162,541]. IDs range among 1, 2, . . . , 162541.
- Movies [62,523]. Each movie has unique IDs and title names. For each movie, one or more genres, like "Mystery" or "Fantasy", are assigned — even if none is assigned, the movie still has a placeholder genre, "(no genres listed)". There are 20 genres, including the placeholder.
- Ratings [25,000,095]. Ratings, provided from the users, are made on a 5-star scale, with half-star increments (0.5 stars - 5.0 stars). Each rating entry also has a timestamp denoting when it was collected.

The tagging data:

- Tags (genome-tags) [1,128]. IDs range among 1, 2, . . . , 1128.
- User-provided taggings (tags) [1,093,360]. Similarly to ratings, these taggings are provided by the users, to reflect users' own views on the movies. They each have a timestamp. And these tagging entries are the original source of all available tags.

- Computer-generated taggings (genome-scores) [13,816 × 1,128]. The GroupLens group did their own research and produced computer-generated taggings for 13,816 movies — for each movie-tag pair, a score was produced to measure their alignment — based on the original user-provided taggings.

For this and the three following chapters, our research will focus exclusively on the base data.

2.2 Feature Engineering Overview

To utilize a machine learning algorithm, raw data usually can't be directly fed as input. Because

- The data is not organized or clean enough to conform to the algorithm's specifications;
- The data's original form is not suitable or optimal to most effectively represent the information for that machine learning problem.

So to clean and wrangle the data and to design an optimal problem-oriented form, we are in effect performing feature engineering. In a Forbes' survey, data scientists spend around 80% of the time on data preparation.

There isn't a formula for every feature engineering problem. The techniques are very specific to the bigger machine learning problem we are trying to solve – it's more like an art. For the following, I'm introducing some common techniques for feature engineering; most of them are also discussed at this web post. **[6]**

The first part, cleaning the data, includes **imputation** and **handling outliers**.

Imputation is about dealing with the missing values. It is reasonable to omit the data point altogether when it doesn't have sufficient features (deleting rows), or to exclude the feature when it only covers a small proportion of the data points (deleting columns). A good threshold of data coverage would be 0.7. But when the missing proportion is small enough, we can still use the data by filling in the missing values ourselves — two common options are a default value (like zero) or an average value (like the median).

For **outliers**, we usually would simply omit them when mining the pattern from the dataset (because they would disproportionately sway the result), but it requires work to detect the outliers. We can use statistical metrics (for example, percentiles and standard deviation) to quickly determine whether a data point is an outlier, but the best way is to sufficiently visualize the dataset and apply human judgment.

Next, I'm now going to introduce four common strategies to transform individual values into features: **one-hot encoding**, **binning**, **log-transformation**, and **scaling**.

One-hot encoding is probably the most common encoding method. It transforms a single categorical data value into a list of 0-or-1 values, one for each category, denoting which category the data point belongs to.

The advantage of such encoding is that the information is split into a list of boolean values or simple counts (e.g. the bag-of-words method), so that category can be independently analyzed and no information is lost. But the obvious disadvantage is that it greatly increased the number of features and the representation complexity.

Binning is about changing the granularity of the data – lumping more finely defined data into broader categories. For example, integers from 0 to 100 can be

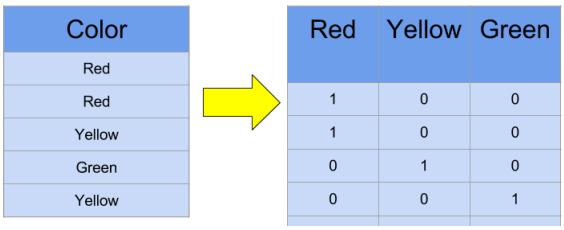


FIGURE 2.1: One-hot encoding example.

binned into three categories: 'low' (0-30), 'medium' (31-70) and 'high' (71-100). And this can be applied to categorical data, too — for example, ethnicity "Chinese" to be re-labeled as "East Asian". The data becomes less granular and therefore more regularized. And the trade-off is just that of regularization — we sacrifice precision to prevent over-fitting.

Log-transformation is simple: just taking the log of the value, and its effect is to "bend the curve" — the value's fluctuation when it's already very large won't matter as much as when the value is small. It places more sensitivity to smaller data than larger data. To avoid situations like "no definition" or negative log values, we add 1 before the logarithm:

$$x \to \log(x+1).$$

Scaling is about controlling the data value's range. There are two common techniques, both are linear transformations. **Normalization** relocates the data, with the minimum to be 0 and the maximum to be 1:

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}.$$

And **(z-score) standardization** extracts the magnitude of deviation in regard to the whole data set, so that the mean μ is now 0 (balanced), and a change of one standard deviation σ in the original data x will result in a change of 1 in the standardized value z:

$$z = \frac{x - \mu}{\sigma}.$$

2.3 My Design for the MovieLens Dataset

First, what is the problem we are trying to solve/answer? Our main concern is to **learn more about the users**. And to start with the base data, it makes sense to compile all the relevant records regarding each particular user into a single profile. This is simple. We will have 4 columns:

- userId
- movieIds (list of movies that user has rated)
- ratings (a list, corresponding to each movie)
- genres (a list, each genre group corresponding to each movie)

And now our concern is to transform this profile into a list of **features** for that particular user.

What can we do immediately? To compute some **statistics** first:

userId	movieIds
27612	8810~912~648~4873~1099~4310~44191~27660~527~4105~48516~45210~5810~5291~6586~6754~3623~33162~3471~30810~1584~2701~8529~1097~33794~45447~2010~4995~4979~2712~7153~34405~5507~2006~45730~783~435~2797~1073~46578~1748~858~145~4874~2959~1206~318~3052~7147~3189~1270~3863~5679~260~8665~4226~3638~5690~3624~1831~208~1233~7099~4015~6934~3617~104~30825~1485~4993~480~50005~1907~293~1380~1779~3969~10~33166~8528~608~39183~1275~6188~53125~2716~2985~45186~2396~1876~597~8376~3751~6297~296~2826~40278~4816~4369~173~1127~3988~2539~457~46530~1215~27728~2115~27773~4018~45722~1320~4148~5782~8644~95~2161~33493~2455~4262~500~5574~6440~594~30793~786~1251~2289~8641~4963~2657~48043~1682~1253~919~2648~34~6365~6264~2915~1580~5872~1356~5881~1221~1917~37729~47~1923~3977~595~1947~1287~355~6863~46976~4299~41997~596~34048~1873~1201~5952~6323~

FIGURE 2.2: User ID and movie compilation.

ratings
3.5~4~4~4.5~4.5~2.5~3.5~4~4.5~4~4~3.5~3~4.5~3~3.5~1~4.5~5~4.5~4.5~1~1.5~4.5~4.5~1~4.5~5~4~3.5~4~4.5~1~3~2~4~3~3.5~4~4~5~4.5~1~4.5~4.5~4.5~4.5~3~4~3~3.5~3.5~4~4.5~3~4~3.5~4.5~3~3.5~4~4.5~4.5~2.5~2.5~2~3~3~3~4~3.5~4~4.5~4.5~3~3.5~3.5~4.5~4~3~4~3.5~4.5~3~3~3.5~4~3.5~4~4~3.5~2.5~2.5~3.5~3.5~3.5~3.5~4~2~3~4.5~3.5~4~4~2.5~4~4~3~3.5~2.5~3.5~3~4~3.5~3~3~4~4~4~4~2.5~2.5~4~3.5~3.5~3.5~5~4~3.5~2.5~4~4.5~3.5~3.5~4.5~4~2~3~3.5~3.5~3.5~3.5~4~3.5~4~4~3.5~3~3~2.5~3.5~3.5~4~3~3.5~3.5~3~4~4~4~4~4~4~4~3~3.5~3~3.5~4~4~4.5~3.5~4~4~3.5~3.5~4.5~3~3.5~4~4~4.5~3~4~4~4~3~3~4~3.5~3~3.5~3~4~3~3~3.5~4~3.5~3.5~3.5~2~2.5~4~4.5~5~4~4~4.5~4~4.5~3.5~4~4.5~4.5~2~2.5~3~4.5~3~3~3.5~4~3.5~3.5~4~3.5~4~4~3.5~3.5~3.5~4.5~2.5~3~3.5~3.5~4~3~4~3.5~3.5~3.5~3.5~4.5~4.5~4.5~3.5~3~3.5~2.5~4.5~4~4.5~3~3~4~3.5~4~2.5~3~3.5~4~3.5~4~4~2~3~4~4~4~4.5~4~3.5~3.5~4~4~4~3.5~4~3.5~3~3.5~2.5~3.5~2.5~4~2.5~3.5~3.

FIGURE 2.3: Rating compilation.

- total_count. Total number of movies the user has rated.
- avg_rating. Average of the all the user's ratings.
- cnt_Mystery, etc. A movie count for each of the 20 genres. Note that because each movie can have multiple genre labels, the sum of all 20 numbers will exceed the total count.
- avg_Mystery, etc. The average rating of all the user's ratings that belong to each particular genre.

Row	userId	total_count	avg_rating	cnt_Mystery	cnt_Fantasy	cnt_Musical	cnt_Documentary	cnt_None
1	121015	86.0	3.5406976744186	7.0	6.0	1.0	1.0	0.0

FIGURE 2.4: User statistics: total_count, avg_rating, cnt_Genre.

Then can we directly use these statistics as features, to feed into the machine learning pipeline? There are two main issues.

The first issue is the data range. total_count can range from 20 to several thousands (the largest count is 32202) and this count dominantly determines the level of other genre counts. The rating statistics will be no larger than 5. And when two users' lists of features are compared together, the count features will overshadow the rating features. We can consider applying weights, but the same issue arises again: when the weights scale down the count features of large values, they trivialize the count features of small values.

The second issue is — even if these features share the same value range – the meaning of each feature and whether it aligns with the problem we're trying to solve. From the total count, we want to know if the user is a heavy rater or not, rather than the exact number of ratings. For example, we can do: 20-100 is "low", 101-500 is "medium", and 501+ is "high". But this approach breaks the continuity into discrete categories — we can think about transforming the total count into a continuous "index" suggesting the degree of the rater's engagement. Similarly,

avg_Mystery	avg_Fantasy	avg_Musical	avg_Documentary	avg_None
4.0	3.66666666666667	3.0	0.5	0.0

FIGURE 2.5: User statistics: avg_Genre.

what does an average rating mean — and what do we want to learn from it? We may want to learn the rater's preferences among different genres. And to cancel out the "rating generosity" of the rater, we may also consider the difference between the genre-specific average rating and the overall average rating.

Therefore we can use these statistics as a basis, and generate features that are more **range-regular** and **meaningful**.

But before we proceed, there is another set of data we can utilize: the **global statistics**. Specifically, the average (or mean) of counts and ratings.

stats_global

Schema Details Preview

Row	row	mean	median
1	Documentary	1.98380101020666	0.0
2	None	0.163817129216628	0.0
3	Film_Noir	1.52101315975662	0.0
4	Musical	5.93236168105278	2.0

FIGURE 2.6: Global statistics: mean and median of the number of rating belonging to each genre.

Then now we can further build the features to serve as input for the next machine learning stage: **count_score**, **interest_score** (for each genre), **avg_rating**, and **div_rating** (for each genre).

The first feature, **count_score**, transforms the total_count into a real number in [0,1], using user-set parameters: **power** and **total_threshold**:

```
power = 0.5

if total_count > total_threshold:
    count_score = 1
else:
    count_score = (total_count / total_threshold) ** power
```

The first if-statement categorizes all raters whose number of ratings exceeds a certain threshold to "fully engaged", with the count_score being 1. The else-statement deals with the level of engagement for "partially engaged" raters and obviously should increase with respect to the total_count. It should also be more sensitive to a smaller total_count, therefore I used the power function with a power smaller than 1. Note that this power, now set as 0.5, can be adjusted later to better suit the problem and the data.

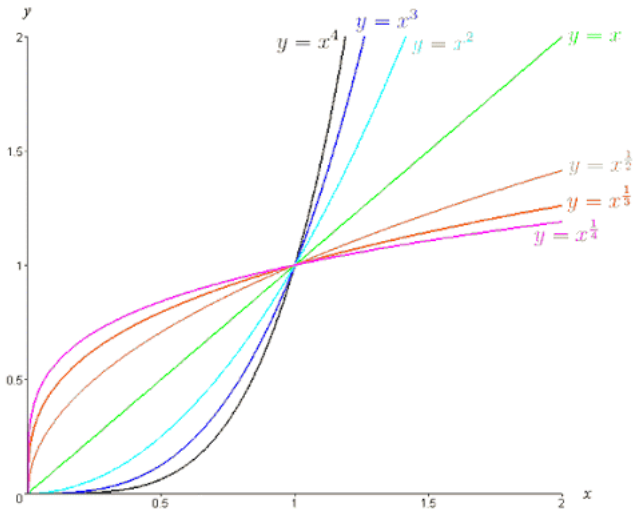


FIGURE 2.7: Power functions with different powers. Taken from reference [12].

Then for each genre (with a genre_count), we compute 2 scores: **in_score** and **ex_score**, and they are defined by the following formula — with user-set parameters: **in_power**, **ex_power** and **genre_threshold**.

```
in_power = 0.5
in_score = (genre_count / total_count) ** in_power

ex_power = 0.5
if genre_count >= genre_threshold:
    ex_score = 1
else:
    ex_score = (genre_count / genre_threshold) ** ex_power
```

in_score represents the user's preference towards this genre, compared to the user's other genre commitments. That's the reason why we're using the fraction over the total_count, and for the same reason of "bending the curve", we are using the power function with a parametric power.

ex_score represents the user's commitment in a more absolute sense. It uses the same formula as the count_score, with parameters genre_threshold and ex_power.

```
in_weight = 0.5
ex_weight = 1 - in_weight
interest_score = (in_score ** in_weight)*(ex_score ** ex_weight)
```

(Genre-specific) **interest_score** is the final product of these two scores — the weighted geometric average of in_score and ex_score. Note that this pair of weights, **in_weight** and **ex_weight**, can be arbitrarily distributed, but should have the sum of 1.

avg_rating is the same statistic we have calculated before, the overall average rating among all ratings the user submitted. And (genre-specific) **div_rating** is the rating surplus (or deficiency) that that genre rating has, compared to the total average:

```
div_rating = genre_rating − avg_rating
```

The **count_score** and **interest_score**s are defined strictly within the interval [0, 1]. And the **ave_rating** is among [0, 5]. And **div_rating**s will theoretically be within [-5, 5], but will most likely be among (-1, 1).

Schema Details Preview

Row	userId	count_score	int_Mystery	int_Fantasy	int_Musical	int_Documentary
1	122866	0.294428605808196	0.394610191805376	0.209147663589187	0.273786973539035	0.0
2	155750	0.294428605808196	0.0	0.0	0.387193251179987	0.0

FIGURE 2.8: Features.

This way we have a list of features that are **meaningful** and generally **evenly bounded**. They also utilize most of the information from the statistics. And for those parameters, we can set them in relation to the global statistics. For example:

```
total_threshold = 0.7 * avg_total_count
genre_threshold = 0.7 * avg_genre_count
```

The effectiveness of these features can be evaluated based on the effectiveness of downstream results, because features are to be designed for the following machine learning steps. That opens the next chapter: Clustering.

Chapter 3

Clustering

3.1 Common Clustering Methods

The goal of clustering is to divide the data sample into groups, such that:

- The data points within the same group should be similar,
- And the data points belonging to different groups should be dissimilar.

Of course, here similarity is something to be specifically defined — and based on different similarity measures we should have different clustering results. Another factor of clustering is the number of clusters. This, in different algorithms, can be either specified beforehand or produced as the algorithm's result.

Why is clustering relevant to our problem — *to understand media users*? Because we do not just want to know everything about individual users (and this is usually practically impossible). We also want to learn how the users form groups among themselves and the characterization of these groups. This grouping can serve as valuable information for the marketing staff, as they can target individuals with just the traits of their membership group (focus groups), with the complexity substantially reduced. (For example, they can recommend movies!) Another benefit for performing clustering is that the associations to similar users and cluster groups are another kind of information, something potentially useful in analyzing the user himself/herself.

Clustering is a classic unsupervised learning problem, with so many approaches theoretically available. Here I'm introducing some of them most relevant to our research context.

3.1.1 K-Means Clustering

This is probably the most commonly used method. **[7]** The nice thing about K-Means is its simplicity and computation efficiency. Note that the user needs to predetermine the number of clusters (K) before the algorithm starts.

Each cluster group has a "cluster center". The algorithm reiterates these two steps:

- **Re-centering**: for each cluster group, update the cluster center as the average of its members. (For the beginning iteration, we can assign the K cluster centers either randomly, or arbitrarily by a default fashion. This is called seeding.)
- **Re-assigning**: reassign the associations between data points and the cluster centers (thus their cluster groups) — each data point now belongs to its closest cluster center.

In the end the K cluster centers should converge — either move only minimally below a certain threshold, or stay put. Note that if the clustering associations don't change after one iteration of the two steps, they will forever stay the same.

Aside from the requirement of determining the cluster number K beforehand, the procedure can also be sensitive to outliers.

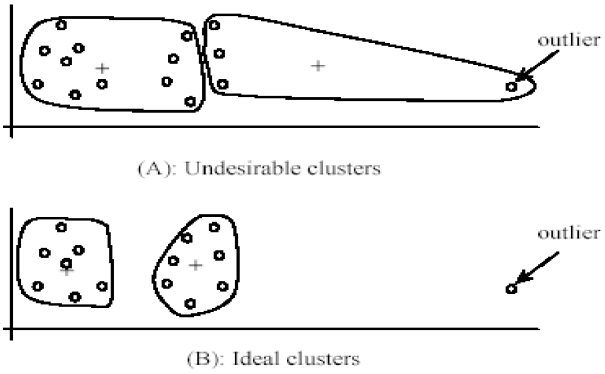


FIGURE 3.1: K-Means: sensitivity to outliers. Taken from reference [8].

3.1.2 Hierarchical Clustering

Hierarchical clustering **[8]** is about performing clustering step by step, each step's result is built on the previous step's. These results eventually form a "hierarchy", and the final clustering will emerge as one intermediate result.

There are two kinds of hierarchical clustering: the **divisive** approach (top-down) and the **agglomerative** approach (bottom-up).

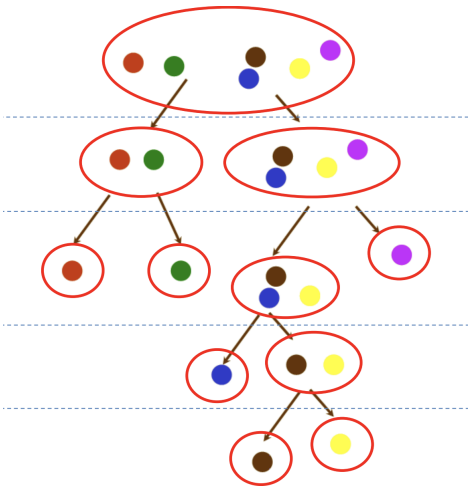


FIGURE 3.2: A hierarchical clustering approach: divisive. Taken from reference [8].

Specifically, in the **divisive** approach, starting from all data members being in one single group, each step takes a division — to break an existing cluster group in half. Usually it would be the "widest present gap", however it is defined. For a

group of N members, it will take $N-1$ steps to produce N groups of isolated single members, i.e. the hierarchical tree will have $N-1$ parent nodes.

The **agglomerative** approach does the opposite — it starts from N groups (N isolated data members) and, at each step, it combines two existing groups, until finally after $N-1$ steps, the entire data sample is grouped into one cluster. It makes sense that at each step we combine the "closest two groups" together, based on certain definition.

Obviously in both approaches, the beginning and end states — all members belonging to the same cluster or each member consisting its own cluster — wouldn't be useful clustering. A good clustering would a step in the middle, and this can be determined *after* the hierarchy has been formed.

With proper definitions of "widest gap" and two "closest groups", both approaches can be deterministic, but the **agglomerative** approach generally takes less time, as it, at each step, compares all pairwise distances among the clusters (at most), whereas the **divisive** approach needs to consider all possible partitions of every cluster at each step.

3.1.3 Fuzzy Clustering

This is a generalization of the K-Mean algorithm. In K-means, there are K cluster centers, and at all times, each data point is associated with one and only one cluster center. In Fuzzy Clustering, we still have K cluster centers (with a predetermined K), but each data point can belong to *all* K clusters, with a set of K weights of its own prescribing the degree of association.

Specifically, we have a data sample $\{x_1, x_2, \ldots, x_n\}$, and the algorithm starts, likewise, with K cluster centers $\{c_1, c_2, \ldots, c_K\}$. We use a weight matrix

$$W = w_{ij} \in [0,1], \;\; i = 1,2,\ldots,n, \;\; j = 1,2,\ldots,K,$$

where w_{ij} describes the degree of association between element x_i and center c_j. (We always have $\sum_j w_{ij} = 1$ for any element x_i.)

In addition, we have a **weight degree** m, also pre-determined, similar to K.

The algorithm still iterates between two steps, **re-assigning** and **re-centering**, but at each step it aims to minimize the following objective function:

$$\sum_{i=1}^{n}\sum_{j=1}^{K} w_{ij}^m ||x_i - c_j||^2.$$

As a result, for the **re-assigning** step, the weights w_{ij} will be determined as (using the Lagrange multiplier method)

$$w_{ij} = \frac{1}{\sum_{k=1}^{K} \left(\frac{||x_i - c_j||}{||x_i - c_k||}\right)^{\frac{2}{m-1}}}.$$

It's easily verifiable that $\sum_{j=1}^{K} w_{ij} = 1$ holds for any i.

And for the **re-centering** step, the cluster centers c_j will be determined as

$$c_j = \frac{\sum_{i=1}^{n} w_{ij} x_i}{\sum_{i=1}^{n} w_{ij}}.$$

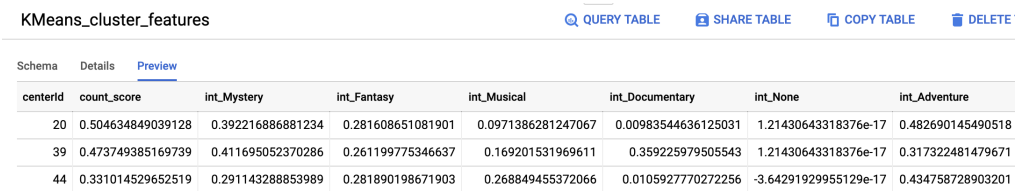
In the end, when both the weights w_{ij} and the cluster centers c_j converge (move below a certain threshold), we've produced a stable fuzzy clustering.

We can now see that the K-Means algorithm is only a special case, where the weights are taken from two values $\{0, 1\}$, whereas in the general case of Fuzzy Clustering weights can be any real numbers in the internal $[0, 1]$.

3.2 My Design for Clustering

In my own experiment, I used simply the K-means algorithm, supplied by the scikit-learn platform. It requires two parameters: number of cluster centers (n_clusters) and an integer specifying the randomness for the initiation of cluster centers (random_state). I randomly selected 10000 users for this cluster analysis, and I set the cluster number to be 50. And I set the random_state to be 0.

As a result, we have 50 centroids of the 10000 user features, numbered as 1, 2, ..., 50. And each user is labeled with a centroid ID.



centerId	count_score	int_Mystery	int_Fantasy	int_Musical	int_Documentary	int_None	int_Adventure
20	0.504634849039128	0.392216886881234	0.281608651081901	0.0971386281247067	0.00983544636125031	1.21430643318376e-17	0.482690145490518
39	0.473749385169739	0.411695052370286	0.261199775346637	0.169201531969611	0.359225979505543	1.21430643318376e-17	0.317322481479671
44	0.331014529652519	0.291143288853989	0.281890198671903	0.268849455372066	0.0105927770272256	-3.64291929955129e-17	0.434758728903201

FIGURE 3.3: K-Means results: centroids of user features.

KMeans_labels

Schema Details Preview

Row	userId	centerId
1	72421	0
2	24919	0
3	68571	0
4	125114	0

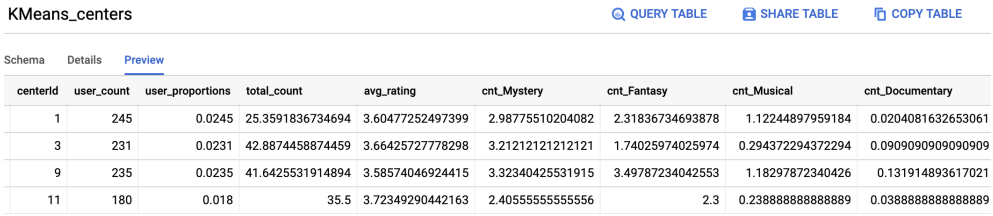
FIGURE 3.4: K-Means results: associations between users and centroids/clusters.

Note that we are performing clustering on the *features* of users, rather than the direct *representations* of users (in our case it's the first-round statistics, used to generate features afterwards). More specifically, for user representation $\mathbf{x}$, his feature is $f(\mathbf{x})$. After performing K-Means on features like $f(\mathbf{x})$, we can have clusters (for example: cluster C containing that user, $\mathbf{x} \in C$), and we have K-Means centroids, such as $c(f(\mathbf{x}))$. But this is not necessarily the feature of the cluster center of C:

$$c(f(\mathbf{x})) \neq f(\mathbf{center}(C))$$

So to get the centers of clusters, the right procedure should be to recalculate the centers $\mathbf{center}(C)$ once we have the cluster groupings, instead of translating back from the feature centroids: $f^{-1}(c(f(\mathbf{x})))$. So we can now do this because we have

the cluster associations (the labels), and the cluster center would just be the mean of all the cluster members.



KMeans_centers

QUERY TABLE SHARE TABLE COPY TABLE

Schema Details Preview

centerId	user_count	user_proportions	total_count	avg_rating	cnt_Mystery	cnt_Fantasy	cnt_Musical	cnt_Documentary
1	245	0.0245	25.3591836734694	3.60477252497399	2.98775510204082	2.31836734693878	1.12244897959184	0.0204081632653061
3	231	0.0231	42.8874458874459	3.66425727778298	3.21212121212121	1.74025974025974	0.294372294372294	0.0909090909090909
9	235	0.0235	41.6425531914894	3.58574046924415	3.32340425531915	3.49787234042553	1.18297872340426	0.131914893617021
11	180	0.018	35.5	3.72349290442163	2.40555555555556	2.3	0.238888888888889	0.0388888888888889

FIGURE 3.5: Cluster centers recalculated after cluster grouping is determined.

This way the cluster centers and the user representations are of the same form. And the next question would be *how to measure the distances* — between the users or between the user and his cluster center?

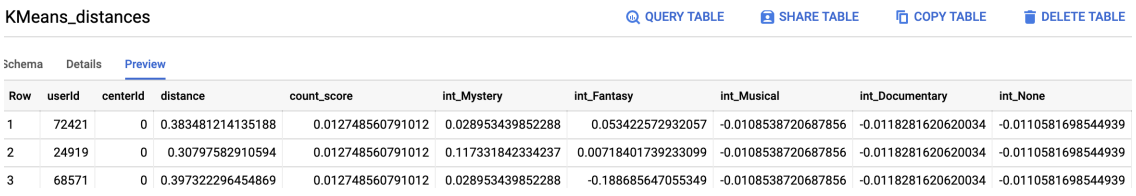
A direct idea would be just the Euclidean distance, to calculate the differences between corresponding coefficients and to accumulate them by summing the squares of the differences and taking the squared root. The issue with this approach is that each coefficient, each *category*, means a different thing. And we need to apply different weights before taking the Euclidean norm. Below are the weights I chose:

```
WEIGHT_CNT = 3.0
WEIGHT_INT = 1.0
WEIGHT_DIV = 0.1

def feature_weight(feature):
    if feature == 'count_score':
        return WEIGHT_CNT
    elif feature.startswith('int'):
        return WEIGHT_INT
    elif feature.startswith('div'):
        return WEIGHT_DIV

WEIGHTS = [feature_weight(feature) for feature in FEATS_LIST]
```

Then we have the following distances calculated, between each user and the corresponding cluster center. Note that only the "distance" column is the final weighted distance, other columns show the individual differences.



KMeans_distances

QUERY TABLE SHARE TABLE COPY TABLE DELETE TABLE

Schema Details Preview

Row	userId	centerId	distance	count_score	int_Mystery	int_Fantasy	int_Musical	int_Documentary	int_None
1	72421	0	0.383481214135188	0.012748560791012	0.028953439852288	0.053422572932057	-0.0108538720687856	-0.0118281620620034	-0.0110581698544939
2	24919	0	0.30797582910594	0.012748560791012	0.117331842334237	0.00718401739233099	-0.0108538720687856	-0.0118281620620034	-0.0110581698544939
3	68571	0	0.397322296454869	0.012748560791012	0.028953439852288	-0.188685647055349	-0.0108538720687856	-0.0118281620620034	-0.0110581698544939

FIGURE 3.6: Distances between each user and the corresponding cluster center.

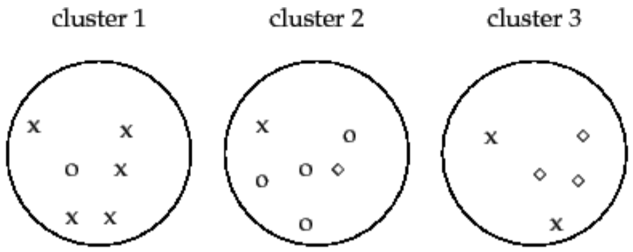
```
difference = user_score − center_score
```

And then we should ask: is this clustering good?

3.3 Evaluation of Clustering

The evaluation of clustering is not a trivial problem, as the goals of clustering are quite qualitative: the intra-cluster elements should be similar/close, while the inter-cluster elements should be different/far apart. And in our problem, the data points don't have "natural" labels — given two data points, we don't know if they should be grouped together or not, without pre-existing labels/categories.

If such labels do exist, i.e. the data sample is partitioned into classes $C = \{c_1, c_2, \ldots, c_J\}$, and the clustering divides the data sample into another partition $\Omega = \{\omega_1, \omega_2, \ldots, \omega_K\}$, we have some statistics to measure up the effectiveness of clustering. The methods below are selected from the section "Evaluation of clustering" from book **[10]**.



► **Figure 16.1** Purity as an external evaluation criterion for cluster quality. Majority class and number of members of the majority class for the three clusters are: x, 5 (cluster 1); o, 4 (cluster 2); and ⋄, 3 (cluster 3). Purity is $(1/17) \times (5+4+3) \approx 0.71$.

FIGURE 3.7: Purity for data with pre-existing classes. Taken from reference [10].

- **Purity.** Each cluster is labeled as its most prevalent class, and purity is the proportion of data points being correctly labeled by its cluster: (N is the size of data sample)

$$\textbf{purity}(\Omega, C) = \frac{1}{N} \sum_k \max_j |\omega_k \cap c_j|.$$

- **Normalized Mutual Information.** NMI is an information-theoretically based measure: (I is mutual information, and H is entropy)

$$\textbf{NMI}(\Omega, C) = \frac{I(\Omega; C)}{(H(\Omega) + H(C))/2}$$

- **The Rand Index.** We want to assign two data points into the same cluster if and only if they share the same label ("the ideal criterion"). Given any two data points, positive/negative denotes whether or not they belong to the same cluster, true/false denotes whether they satisfy the ideal criterion. Then the pair belongs to one of the four groups: TP, TN, FP and FN. The Rand Index is the proportion of the "true" pairs:

$$\textbf{RI}(\Omega, C) = \frac{\textbf{TP} + \textbf{TN}}{\textbf{TP} + \textbf{TN} + \textbf{FP} + \textbf{FN}}.$$

- **The F Measure.** The F Measure ($\textbf{F}_\beta$) is a weighted average of precision (**P**) and recall (**R**), determined by a parameter $\beta > 1$:

$$\mathbf{P} = \frac{\mathbf{TP}}{\mathbf{TP} + \mathbf{FP}}, \quad \mathbf{R} = \frac{\mathbf{TP}}{\mathbf{TP} + \mathbf{FN}}, \quad \mathbf{F}_\beta(\Omega, C) = \frac{(\beta^2 + 1)\mathbf{P} \cdot \mathbf{R}}{\beta^2 \mathbf{P} + \mathbf{R}}.$$

Then what about our case, where the class labels C don't exist, and we can only measure the distances among data points (users) and cluster centers?

We can propose two measures, corresponding to the original goals of clustering:

- **Cluster Radius.** Within a cluster, we calculate the distances between the users and the cluster center. The radius can be the root mean square of all the distances (This more heavily penalizes the large distances).
- **Inter-Cluster Distances.** Between clusters, we calculate the distances between pairs of cluster centers. And eventually we can calculate, from each cluster center, the minimal distance to any other cluster centers, and the overall inter-cluster distance (some average of the minimal distances).

3.4 Results of Evaluation

KMeans_cluster_radius

Schema Details Preview

Row	centerId	cluster_size	max_distance	ave_distance
1	41	257	0.864121252695251	0.483766684550974
2	12	258	0.816495274416941	0.489677561006525
3	7	260	0.745110755203241	0.381434452874822
4	49	261	0.792588861908089	0.427848747231632
5	24	275	0.747400835255139	0.43978072801259
6	21	280	0.795043968294745	0.455227051896281
7	37	281	0.70790051874277	0.493912456491646
8	48	294	0.810113804064077	0.479988092720647
9	14	61	1.06675161494184	0.697535355453477
10	19	65	1.05175063538898	0.632807166860715
11	36	83	0.917006066793975	0.576574581723035
12	4	340	0.696856575602807	0.437939482766665
13	44	342	0.761515368960803	0.389620451620545
14	42	87	0.877983971176066	0.571260825428907
15	35	122	0.87552496555714	0.580004253566291
16	26	124	0.791328421126689	0.521952877938059
17	8	132	0.819333265157557	0.514209409340074

FIGURE 3.8: Cluster radiuses: distances within clusters

I used the aforementioned **weighted distance** to measure the difference between two feature rows (users or clusters), for 10000 users and, after using K-Means provided by scikit-learn, 50 clusters. The table above shows a sample — within each cluster, the maximum user distance from the cluster center and the **cluster radius** (average user distance, taken by the root mean square).

3.5 Interpretation of Clusters

After we've done with clustering, one question is to *interpret* the clusters. What do the data points in the same cluster have in common? This can be simple statistics, or some further construction.

cluster_interpret

QUERY TABLE SHARE TABLE COPY TABLE DELETE TABLE EXPORT

Schema Details Preview

Row	center_index	user_percentage	user_count	top_genres	top_genre_scores
1	24	2.0	197.0	['Action', 'Drama', 'Thriller', 'Adventure', 'Crime', 'Sci_Fi', 'Comedy', 'Mystery']	[0.5, 0.45, 0.43, 0.28, 0.27, 0.24, 0.19, 0.1]
2	8	2.0	204.0	['Drama', 'Comedy', 'Romance', 'Thriller', 'Crime']	[0.63, 0.49, 0.32, 0.13, 0.13]
3	25	2.0	198.0	['Comedy', 'Drama', 'Romance', 'Thriller', 'Action', 'Adventure', 'Crime']	[0.57, 0.39, 0.22, 0.19, 0.18, 0.15, 0.14]
4	16	4.0	401.0	['Drama', 'Comedy', 'Romance', 'Thriller', 'Crime', 'Action', 'Adventure']	[0.61, 0.35, 0.22, 0.2, 0.16, 0.14, 0.13]
5	21	4.0	404.0	['Drama', 'Comedy', 'Thriller', 'Action', 'Adventure', 'Romance', 'Crime', 'Sci_Fi', 'Mystery']	[0.56, 0.28, 0.27, 0.24, 0.2, 0.19, 0.18, 0.17, 0.1]
6	4	1.5	148.0	['Action', 'Sci_Fi', 'Thriller', 'Drama', 'Adventure', 'Comedy', 'Crime', 'Horror', 'Mystery', 'Fantasy']	[0.46, 0.44, 0.37, 0.35, 0.32, 0.21, 0.15, 0.13, 0.11, 0.11]
7	0	2.9	286.0	['Drama', 'Comedy', 'Thriller', 'Crime', 'Action', 'Romance', 'Adventure', 'Mystery', 'Sci_Fi']	[0.53, 0.36, 0.33, 0.26, 0.21, 0.19, 0.14, 0.11, 0.1]
8	10	0.6	61.0	['Horror', 'Thriller', 'Drama', 'Action', 'Comedy', 'Sci_Fi', 'Adventure', 'Mystery', 'Crime', 'Fantasy']	[0.59, 0.43, 0.25, 0.24, 0.22, 0.21, 0.13, 0.12, 0.11, 0.11]
9	45	0.6	65.0	['Sci_Fi', 'Action', 'Adventure', 'Thriller', 'Drama', 'Comedy', 'Horror']	[0.67, 0.51, 0.38, 0.34, 0.26, 0.23, 0.13]
10	34	1.2	125.0	['Action', 'Thriller', 'Adventure', 'Comedy', 'Sci_Fi', 'Drama', 'Romance', 'Fantasy', 'Crime']	[0.49, 0.48, 0.44, 0.38, 0.2, 0.18, 0.17, 0.13, 0.12]
11	22	1.2	124.0	['Comedy', 'Drama', 'Children', 'Adventure', 'Romance', 'Fantasy', 'Animation', 'Action', 'Musical', 'Thriller']	[0.51, 0.34, 0.32, 0.32, 0.26, 0.22, 0.21, 0.16, 0.16, 0.12]
12	38	1.2	123.0	['Comedy', 'Thriller', 'Adventure', 'Action', 'Romance', 'Drama', 'Sci_Fi', 'Fantasy', 'Children']	[0.44, 0.37, 0.34, 0.34, 0.26, 0.25, 0.16, 0.14, 0.13]

FIGURE 3.9: Examples of clusters: their interpretation.

The table above shows my interpretation of the 50 clusters produced. It has the number of its users and the user percentage, and also each cluster's **top genre** interests, each characterized by a **genre score**.

Genre scores in this case are simple: just the frequency of the genre showing up in the "cluster center"'s viewing record. (Note the cluster centers are not real users; they are the average of its users, but, because they are of the same representation form as the real users, they can be treated just like users.) Also, because each movie can have multiple genre tags, after all the genre tags have been counted in the frequency, the sum of these genre frequencies can be larger than 1.

In the above example, the top genres are selected when they have a score larger than a **genre threshold**, set as 0.1 in this case.

Chapter 4

Recommendations

4.1 Main Approaches of Recommender Systems

Now that we already have the clustering divisions, we can think about how to utilize that information. One direct application is the recommender systems — recommending movies for users based on that user's and all other people's viewing records. This is a classic machine learning problem, with much mature methods available. From 2006 to 2009, Netflix held a competition for the best recommendation system using its own real user data, with the prize being 1 million dollars.

There are three main approaches to recommender systems: **Content-Based**, **Collaborative Filtering**, and **Latent Factors**. These methods are discussed in the Stanford Course, Mining of Massive Datasets **[11]**, while the following is my own summarization.

Before we dive into the introduction, let's frame the problem in a formal way:

Let X be the set of users, S the set of items (in our case, movies), K the set of ratings, and we have a **utility function** $r : X \times S \to K$, denoting each user's rating for each item, and a **utility matrix** $R = [r(x,i)]_{x,i}$. Some of values in R are present; our job is to predict the missing values.

4.1.1 Content-Based Recommenders

This approach digs down into the *content* of both the items and the users. We build *profiles* for them and then match the closest user-item pairs.

We first build item profiles, but what is a profile exactly? It would be a feature vector, and each feature represents an descriptive aspect of the items. For example, for movies, these features can be author, title, cast, staff, production year, etc. And for text documents, the features can be counts of "important words", pre-determined by the corpus content.

When we have these item profiles i, we can then build a profile x for each user — to be an average of these items that the user has rated.

Intuitively, it shouldn't be the simple mean, because not all items should matter the same. Then we can use again the weighted mean, with the weight being, conveniently, the user's rating for that item. Or the weight could be some further processing of the rating. For example, the weight to be the *deviation* of that rating, compared to the average level of users' rating level.

Then with i and x available, how can we determine the compatibility, or *similarity*? We can consider the following formula:

$$u(x,i) = \cos(x,i) = \frac{x \cdot i}{||x|| \cdot ||i||}.$$

The **cosine similarity** can be used for prediction.

The advantage for this content-based approach is that it can immediately determine the potential fondness towards a new item before it is rated by any user, and that it provides an explanation for the prediction. The downside is — how do we recommend for a new *user*? When he doesn't have a viewing record and we can't build a profile for him. And this approach will solely base the recommendations on the user's viewing history, what he has *already* liked, without new possibilities to discover new unknown interests.

4.1.2 Collaborative Filtering

Collaborative filtering uses *parallel, similar* records to predict missing ratings and make new recommendations. It doesn't need to decompose a single rating; instead, it views them as atomic building blocks. Specifically, we have the utility matrix R for all the user-item ratings, and to predict new ratings for user x (a row), we need to find other N *similar* users (rows) with respect to x and use this collection of records to supply missing values in x's row. Here N is a pre-determined parameter.

How do you define "similar"? Here are a few options:

- **Jaccard Similarity.** It simply counts the overlap of presence between two users' viewings, ignoring the ratings for them. For users x and y, the sets of their rated items are X and Y, then their Jaccard similarity is

$$sim_J(x,y) = \frac{|X \cap Y|}{|X \cup Y|}.$$

  Apparently, this omits some information, but in another sense, this omission makes sense: the user may have the same amount of interest towards the items he's rated, regardless of the ratings.

- **Cosine Similarity.** If the two rows for users x and y in U are r_x and r_y, then

$$sim_{cos}(x,y) = \cos(r_x, r_y) = \frac{r_x \cdot r_y}{||r_x|| \cdot ||r_y||}.$$

  For the missing values we fill in zeros.

  The issue is that the missing values have even more of negative impact than the low-rated items — if the two users only share a small overlap of rated items.

- **Pearson Correlation Coefficient.** This is the most comprehensive design of the three. For users x and y, we have r_x and r_y to be their matrix rows, $\bar{r}_x$ and $\bar{r}_y$ to be their rating average, and C to be the set of common items they have both rated. Then we transform the two rating rows r_x and r_y: $\hat{r}_x$ only includes all the ratings of r_x for items in the set C, but for each rating $r_{x,i}$ we modify it as:

$$\hat{r}_{x,i} = r_{x,i} - \bar{r}_x$$

  Then we have new (shortened) rating rows $\hat{r}_x$ and $\hat{r}_y$. Then the Pearson Correlation Coefficient similarity would be:

$$sim_{PCC}(x,y) = \cos(\hat{r}_x, \hat{r}_y) = \frac{\hat{r}_x \cdot \hat{r}_y}{||\hat{r}_x|| \cdot ||\hat{r}_y||}.$$

This design eliminates the influence of missing values and also is adjusted according to the levels of rating generosity of the users.

So now when we have the definition of similarity, for each user x and item i, we can find the N neighbors closest to u who have rated item i. We define these neighbor users form a set $N(x, i)$. The last question becomes: how to synthesize all of the neighbors' ratings for rating prediction $\hat{r}_{x,i}$?

Again there are two options are available: we can find the simple average of these neighbors or the weighted average with the similarities we calculated being the weights.

The interesting thing about collaborative filtering is that the two dimensions of the utility matrix, "user" and "item", are symmetrical. We can calculate similarity between users, so can we perform the same analysis between items. And in fact, usually, this actually works better — two items are more easily similar than two users on the utility matrix. Because users may be commonly interested in the same item, but they may have different *compositions* of interests, so it's harder to find two users similar as a whole.

And we can also have some more nuanced adjustment to the synthesis step, aside from adopting the item-item similarity approach.

For each user-item pair (x, i), we can at first calculate a baseline rating $b_{x,i}$ (even if the rating $r_{x,i}$ is present):

$$b_{x,i} = \mu + b_x + b_i,$$

where μ is the overall average of all ratings, b_x is the *deviance* of user x (the difference between his average rating over the total average, $b_x = \bar{r}_x - \mu$), and b_i is the *deviance* of item i, similarly defined.

And then we can use the item-item collaborative filtering to add *adjustment*:

$$\hat{r}_{x,i} = b_{x,i} + \frac{\sum_{j \in N(i,x)} s_{ij}(r_{x,i} - b_{x,j})}{\sum_{j \in N(i,x)} s_{ij}},$$

where $N(i, x)$ is a set of N items most similar to i among the rated items of x, and s_{ij} is item-item similarity.

If there aren't enough similar items in user x's rating history, we can switch back to the user-user collaborative filtering pattern.

Compared to content-based approach, the collaborative filter approach doesn't need to understand deeper into "why" users would like items, or it is particularly useful when the innate content analysis cannot generate enough information to make nuanced recommendations. But this approach heavily depends on the **density** of the utility matrix, so for users with very unique taste or unpopular items, or for new users and newly introduced items, it can be very hard to apply collaborative filtering.

And maybe in those cases, we can go back to the content-based recommendations. Or to design a model to combine the two recommendations together — content-based approach's **cosine similarity** and collaborative filtering's **predicted rating** can be two quantifying scores to be combined together.

4.1.3 Latent Factors

The **"Latent Factors"** approach tries to decompose the utility matrix R into two matrices P and Q:

$$R = Q \cdot P^T.$$

But what does this mean? And how would it be helpful?

P is a factorization of items, and Q is a factorization of users. Similar to the content-based approach, "factors" are just features describing the items and users, except that they also need to "work together" and satisfy a hypothesis — the decomposition of ratings:

$$r_{x,i} = q_x \cdot p_i^T = \sum_{f \in F} q_{x,f} p_{i,f}$$

where, $R = [r_{x,i}]_{x \in X, i \in S}$ is our utility matrix, F is the set of factors, and $P = [p_i]_{i \in S} = [p_{i,f}]_{i \in S, f \in F}$ and $Q = [q_x]_{x \in X} = [q_{x,f}]_{x \in X, f \in F}$ are the factorization matrices for items and users.

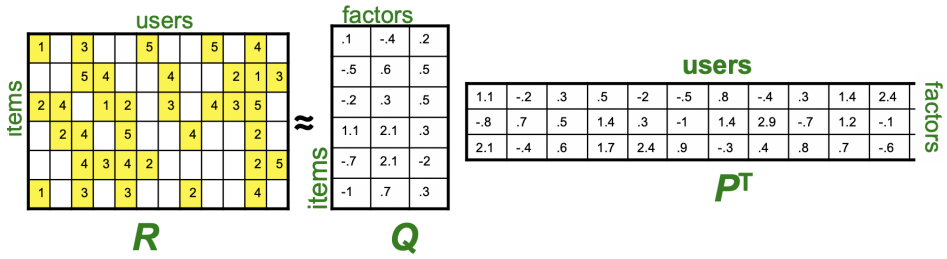


FIGURE 4.1: Decomposition of the utility matrix. Taken from reference [11].

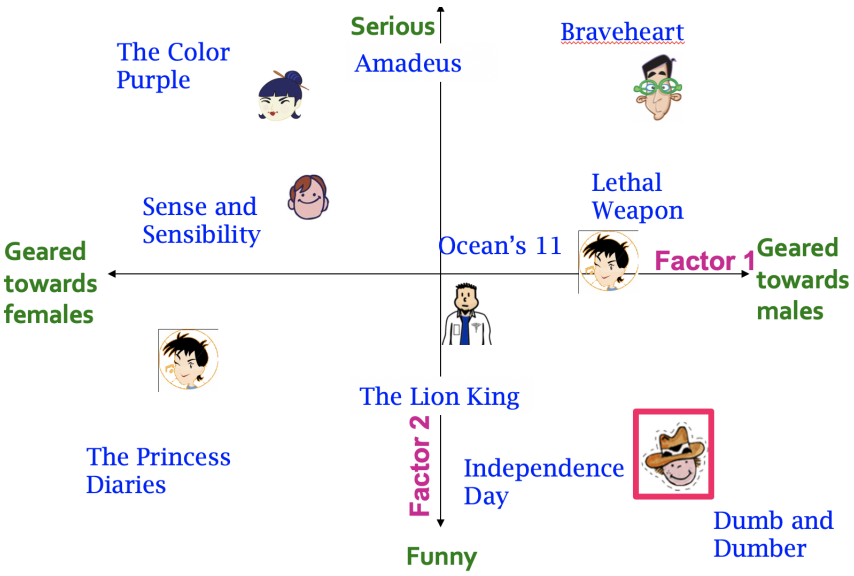


FIGURE 4.2: Example: two factors for movie items. Taken from reference [11].

The usefulness is that if we can use the present values in R to determine P and Q according to the hypothesis equation, then by $Q \cdot P^T = R$ we can map out the full R, thus predicting the missing ratings in R.

And we can transform this problem into an optimization one:

$$P, Q = \textbf{argmin}_{P,Q} \sum_{r_{x,i} \in R} (q_x \cdot p_i^T - r_{x,i})^2 \quad (P = [p_i]_{i \in S}, Q = [q_x]_{x \in X})$$

The issue, of course, is over-fitting, as the matrix factorization can be very sensitive to the outlier rating values. So to add **regularization**, it's more reasonable to minimize:

$$P, Q = \textbf{argmin}_{P,Q} \sum_{r_{x,i} \in R} (q_x \cdot p_i^T - r_{x,i})^2 + \lambda_1 ||p_i||^2 + \lambda_2 ||q_x||^2.$$

And how to achieve this? One option is **stochastic gradient descent**.

The vulnerability of thus approach is *the hypothesis*, obviously. It assumes the innate decomposition of ratings, but if this linear structure is completely senseless with regard to our data, the "decomposition" can be nothing more than misleading. It requires extensive testing to confirm validity.

4.2 My Design and Experiments

I went along with the content-based approach, except I'm recommending movies for the *clusters*, instead of individual users. This has a benefit — it combines the interest of similar users, instead of one user alone.

cluster_recs

Schema Details Preview

Row	centerId	movieId	affinity	title	genres
1	18.0	154676	0.78	Dad (2010)	Drama
2	18.0	173875	0.78	AWOL (2016)	Drama
3	18.0	117452	0.78	Tied (2013)	Drama
4	18.0	88321	0.78	Betty (1992)	Drama
5	18.0	137351	0.78	Fiona (1998)	Drama
6	18.0	121362	0.78	Oviri (1986)	Drama
7	18.0	86583	0.78	Storm (2009)	Drama
8	18.0	4762	0.78	Djomeh (2000)	Drama
9	18.0	5318	0.78	Joshua (2002)	Drama
10	18.0	153046	0.78	Page 3 (2005)	Drama
11	18.0	156377	0.78	Strays (1997)	Drama

FIGURE 4.3: Recommendations for clusters.

Specifically, for 50 clusters, I selected 1000 movies randomly, and designed an **affinity score** that characterizes the alignment between movies and clusters to make

these recommendation decisions — the method "rec_decision" takes in a feature row of cluster center and one of a movie and return a boolean (whether to recommend) and an affinity score (how strong is this recommendation):

```
def rec_decision(center_row, movie_row):
    # some code to extract the top genre scores of clusters
    movie_genre_scores = pass

    interesting = False
    for score in movie_genre_scores:
        if score >= GENRE_THRESHOLD:
            interesting = True
    if interesting:
        affinity = sum(movie_genre_scores)
        if affinity >= AFFINITY_THRESHOLD:
            return True, affinity
        else:
            return False, affinity
    else:
        return False, 0
```

Two parameters, "GENRE_THRESHOLD" and "AFFINITY_THRESHOLD", are preset and adjustable, for this decision making. They were set as 0.10 and 0.70, respectively, in this experiment.

The "AFFINITY_THRESHOLD" serves more like a filter — only selecting the movies that reaches this affinity score, whereas the "GENRE_THRESHOLD" is more influential in the affinity calculation. "GENRE_THRESHOLD" is meant to be a noise filter — only those significant genre interest would be counted in the affinity scoring, excluding the low score points. And with "AFFINITY_THRESHOLD" being 0.70, there were 10094 movie-cluster pairs recommended out of 50000 possible pairs. The top affinity score was 2.46. Even though more than 20% of selected pairs are selected, for each cluster we can still pick just a few with the topmost affinity scores.

This approach has a bias, that is: when a movie is tagged with multiple genres, it is more likely to be recommended than movies with a single genre. And in reality maybe it sometimes should be put as less a priority? For example, if a user likes to watch sports, a movie solely categorized as sports content may be more interesting to that user, than a movie is relevant to sports but also to many other topics. So it is more complicated than a ready solution: normalizing each movie's "genre contribution" to be the same. Because in one scenario, a multi-faceted movie can appeal to many different types of users, but in another one, that movie may feel less concentrated and would not exert a high impact on relevant users.

A different approach is collaborative filtering, apparently. It would be more *focused*, as the "similar collaborators" would only watch a small number from the movie pool. But it would make bring all the limitations of collaborative filtering, as we have discussed — specifically, for users with limited viewing records.

So a **combined approach** can eventually be a better solution — to use clustering as collaborative filtering groups (or to perform the original similarity search), and within the repertoire of all the viewings in a cluster, use the content-based approach to select recommendations for users according to their individual taste/history.

Chapter 5

Conclusion

5.1 Wrapping Up: the Feature Engineering Paradigm

What's so common about the three stages — Feature Engineering, Clustering, Recommendations — is that all three stages are quite **open-ended, unsupervised** learning problems. (Maybe except for Recommendations, we can hide away some *ratings* as the test set and use the rest for training, but in real business cases, such ratings are quite hard to collect and often irrelevant; the vast pool of recommendable options makes it resemble more an unsupervised learning issue.) How to address this nature of these problems?

One aspect is to make sure the methods we use **make sense**, i.e. there is some rationale aligned with our data mining goals and data realities under the mathematical/engineering designs of the model. When designing features, not only do we care about the technical compatibility of these features, we also care about *what these features mean* — the meaning determines their design and their usefulness. The same story goes with Clustering — what do these users clustered together have in common? Are they really similar? What are the clusters used for? And with Recommendations — why are we recommending items to users? Because we have certain marketing goals, want to help users discover new tastes, or just reinforce users' strongest interests? These are not just technical problems, e.g. looking for the best, most complex algorithm to optimize a numerical metric — they require the algorithms to be **understandable** and, in many cases, **flexible for changes**. This is why many methods presented are no more complicated than some high school math, but when applied appropriately, they can produce good results.

Another aspect is **iterative revisions**, to make sure the model works well with the reality, even if we think we have understood all the reasoning behind the model. Because ultimately, these rationales are no more than hypotheses (for example, the Latent Factor method in Chapter 4). Sky's data then has a natural advantage in this aspect, because they constantly collect users' current viewing activities, whereas some MovieLens data was collected more than a decade ago, and its reliability to reflect the users' taste is questionable. For Sky, they can initially set up a simple, crude model, but if they have frequent data updates and analysis, this iteration probably will provide insights no less than the thoughts put in the initial design. They can **build the model iteratively**.

5.2 An Alternative: the Deep Learning Paradigm

An alternative approach, mentioned in the beginning chapter, is the **Deep Learning Paradigm**. This approach only does minimal data processing and delegate the end-to-end hard work (for example, from raw ratings to recommendations) to the neural

net architecture. It has the potential to discover unexpected, or unexpressible, data patterns and can be more convenient and computationally powerful. But the trade-off is the explanability — the deep learning approach is far more opaque — and thus this may make the approach less reliable. As a consequence, we need extensive testing before deploying such a deep learning model.

5.3 Looking Ahead: User Habits

Much of the thesis deals with the problem of **user tastes**, i.e. what content do users like or take an interest in? But another dimension of data would be **timing**, which corresponds to the problem of **user habits** — how do they spend time in their viewings? Are there correlations between timing and genre, habit and taste? Like the genre tags, what do the timing records say about the users? Sky's data is uniquely positioned to address this, given it has the start time of each customer activity. MovieLens data, however, can't provide much more information on users' *viewing habits*, as the timestamps are of when users submitted the ratings, rather than watched the movies.

Appendix A

Bibliography

1. *Sky UK.* https://en.wikipedia.org/wiki/Sky_UK
2. *Data preprocessing for machine learning: options and recommendations.* https://cloud.google.com/solutions/machine-learning/data-preprocessing-for-ml-with-tf-transform-pt1
3. *MovieLens 25M Dataset.* https://grouplens.org/datasets/movielens/25m/
4. F. Maxwell Harper, and Joseph A. Konstan. *The MovieLens Datasets: History and Context.* ACM Transactions on Interactive Intelligent Systems, December 2015, Article No.: 19. https://doi.org/10.1145/2827872
5. Jesse Vig, Shilad Sen, and John Riedl. 2012. *The Tag Genome: Encoding Community Knowledge to Support Novel Interaction.* ACM Trans. Interact. Intell. Syst. 2, 3: 13:1–13:44. https://doi.org/10.1145/2362394.2362395
6. Emre Rençberoğlu. *Fundamental Techniques of Feature Engineering for Machine Learning.* https://towardsdatascience.com/feature-engineering-for-machine-learning-3a5e293a5114
7. *k-means clustering.* https://en.wikipedia.org/wiki/K-means_clustering
8. Shimon Ullman, Tomaso Poggio, Danny Harari, Daneil Zysman, and Darren Seibert. *Unsupervised Learning: Clustering.* http://www.mit.edu/ 9.54/fall14/slides/Class13.pdf
9. *Fuzzy clustering.* https://en.wikipedia.org/wiki/Fuzzy_clustering: :text=Fuzzy%20clustering%20(also%20referred%20to,to%20more%20than%20one%20cluster.
10. Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* Cambridge University Press. 2008.
11. Jure Leskovec, Anand Rajaraman, and Jeff Ullman. *Mining of Massive Datasets.* http://www.mmds.org/
12. http://www.biology.arizona.edu/biomath/tutorials/Power/GraphingPowerFunctions.html