University of Massachusetts, Amherst
Department of Electrical and Computer Engineering
DSP Project

"Machine Learning Models to Build a Recommendation System"

Submitted by:
Shivakumar Valpady
Tejas Panambur

CHAPTER 1: INTRODUCTION

A recommendation system is a computer program that helps a user discover products and content by predicting the user's rating of each item and showing them the items that they would rate highly. Recommendation systems are everywhere. If you've ever looked for books on Amazon or browsed through posts on Facebook, you've used the recommendation system without even knowing it. With online shopping, consumers have nearly infinite choices. No one has enough time to try every product for sale. Recommendation systems play an important role in helping users find products and content they care about.

Often termed as Recommender Systems, they are simple algorithms which aim to provide the most relevant and accurate items to the user by filtering useful stuff from of a huge pool of information base. Recommendation engines discovers data patterns in the data set by learning consumers choices and produces the outcomes that co-relates to their needs and interests.

What can one do with the Recommendation system?

Recommendation systems have several different uses. The most common use for a recommendation system is ranking products by how much a user would like them. If a user is browsing or searching for products, we want to show them the products they would like most first in the list.

Recommendation systems can also be used to find out how similar different products are to each other. If products are very similar to each other, they might appeal to the same users.

Product similarity is especially useful in cases where we don't know much about a particular user yet. We can recommend similar products, even if the user hasn't entered any of their own product reviews yet. We can also use recommendation systems to figure out if two different users are similar to each other. If two users have similar preferences for products, we can assume they have similar interests. For example, a social network can use this information to suggest the two users should become friends.

Awesome usage of recommendation systems

You've probably seen recommendation systems in action on e-commerce websites. When you buy something on Amazon, the next time you visit you will see recommended products based on

your purchase. This is powered by a recommendation system. But that's just the tip of the iceberg. Social media websites like Facebook and Instagram rely heavily on recommendation systems. These websites use recommendation systems to decide which post to display in your timeline and which new friends to recommend to you.

Netflix uses a recommendation system to decide which movies and tv shows to present to you. They are famous for their research and recommendation systems. In 2006 they started the Netflix prize which was a contest for the first team that can improve the recommendation accuracy by 10% would win one million dollars. Three years later the challenge was completed and the prize was awarded.

Recommendation systems also pop up in all kinds of other products. Online dating applications use recommendation systems to decide which users to show to each other. Banks and investment companies use recommendation systems to match different accounts and services to users. Insurance companies do the same.

CHAPTER 2: THEORETICAL EXPLANATION

What are the different types of recommendations?

There are basically four important types of recommendation engines:

- Collaborative filtering
- Content-Based Filtering
- Knowledge-Based systems
- Hybrid Recommendation Systems

Collaborative filtering

This filtering method is usually based on collecting and analyzing information on user's behaviors, their activities or preferences and predicting what they will like based on the similarity with other users. A key advantage of the collaborative filtering approach is that it does not rely on machine analyzable content and thus it is capable of accurately recommending complex items such as movies without requiring an "understanding" of the item itself. Collaborative filtering is based on the assumption that people who agreed in the past will agree

in the future, and that they will like similar kinds of items as they liked in the past. For example, if a person A likes item 1, 2, 3 and B like 2,3,4 then they have similar interests and A should like item 4 and B should like item 1.

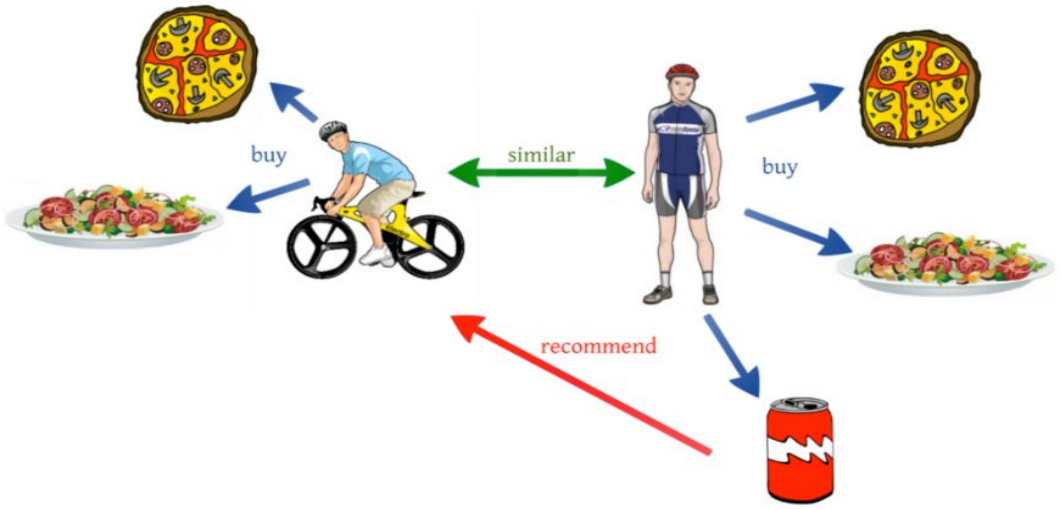

Fig 2.1 : Collaborative Filtering

Further, there are several types of collaborative filtering algorithms:

- **User-User Collaborative filtering:** Here, we try to search for lookalike customers and offer products based on what his/her lookalike has chosen. This algorithm is very effective but takes a lot of time and resources. This type of filtering requires computing every customer pair information which takes time. So, for big base platforms, this algorithm is hard to put in place.

The prediction $\hat{r}_{ui}$ in user based collaborative filtering is :

$$\hat{r}_{ui} = \mu_u + \frac{\sum_{v \in N_i^k(u)} sim(u,v)\,(r_{vi} - \mu_v)}{\sum_{v \in N_i^k(u)} sim(u,v)}$$

Where $\hat{r}_{ui}$ is the estimated rating of user u for item i, μ is the mean of all ratings, μ_u is the mean of all ratings given by user u, $N_i^k(u)$ is the k nearest neighbors of user u that have rated item i and similarity(u,v) is given by:

$$sim(u,v) = \frac{\sum (r_{ui} - \hat{r}_u)(r_{vi} - \hat{r}_v)}{\sqrt{\sum (r_{ui} - \hat{r}_u)^2}\sqrt{\sum (r_{vi} - \hat{r}_v)^2}}$$

- **Item-Item Collaborative filtering:** It is very similar to the previous algorithm, but instead of finding a customer look alike, we try finding item look alike. Once we have item look alike matrix, we can easily recommend alike items to a customer who has purchased any item from the store. This algorithm requires far fewer resources than user-user collaborative filtering. Hence, for a new customer, the algorithm takes far lesser time than user-user collaborate as we don't need all similarity scores between customers. Amazon uses this approach in its recommendation engine to show related products which boost sales.

$$\hat{r}_{ui} = \mu_i + \frac{\sum_{j \in N_u^k(i)} sim(i,j)(r_{uj} - \mu_j)}{\sum_{v \in N_u^k(i)} sim(i,j)}$$

Advantages of item-based filtering over user-based filtering :

- **Scales Better :** User-based filtering does not scale well as user likes/interests may change frequently. Hence, the recommendation needs to be re-trained frequently.
- **Computationally Cheaper :** In many cases, there are way more users than items. It makes sense to use item-based filtering in this case.

Content-based filtering

These filtering methods are based on the description of an item and a profile of the user's preferred choices. In a content-based recommendation system, keywords are used to describe the items; besides, a user profile is built to state the type of item this user likes. In other words, the algorithms try to recommend products which are similar to the ones that a user has liked in the

past. The idea of content-based filtering is that if you like an item you will also like a 'similar' item. For example, when we are recommending the same kind of item like a movie or song recommendation. This approach has its roots in information retrieval and information filtering research.

A major issue with content-based filtering is whether the system is able to learn user preferences from users actions about one content source and replicate them across other different content types. When the system is limited to recommending the content of the same type as the user is already using, the value from the recommendation system is significantly less when other content types from other services can be recommended. For example, recommending news articles based on browsing of news is useful, but wouldn't it be much more useful when music, videos from different services can be recommended based on the news browsing.

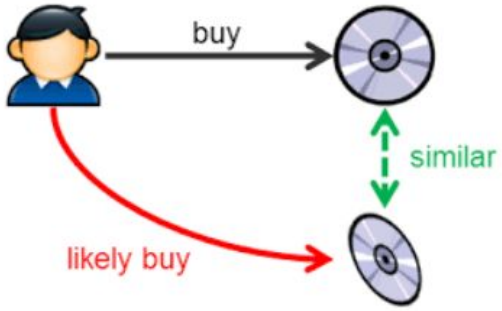

Fig 2.2: Content Based Filtering

Knowledge-Based Recommendation systems

Knowledge-based recommenders are used for items that are very rarely bought. It is simply impossible to recommend such items based on past purchasing activity or by building a user profile. Take real estate, for instance. Real estate is usually a once-in-a-lifetime purchase for a family. It is not possible to have a history of real estate purchases for existing users to leverage into a collaborative filter, nor is it always feasible to ask a user their real estate purchase history.

In such cases, you build a system that asks for certain specifics and preferences and then provides recommendations that satisfy those aforementioned conditions. In the real estate example, for instance, you could ask the user about their requirements for a house, such as its locality, their budget, the number of rooms, and the number of storeys, and so on. Based on this information, you can then recommend properties that will satisfy all of the above conditions.

Knowledge-based recommenders also suffer from the problem of low novelty, however. Users know full-well what to expect from the results and are seldom taken by surprise.

Hybrid Recommendation systems

Recent research shows that combining collaborative and content-based recommendation can be more effective. Hybrid approaches can be implemented by making content-based and collaborative-based predictions separately and then combining them. Further, by adding content-based capabilities to a collaborative-based approach and vice versa; or by unifying the approaches into one model.

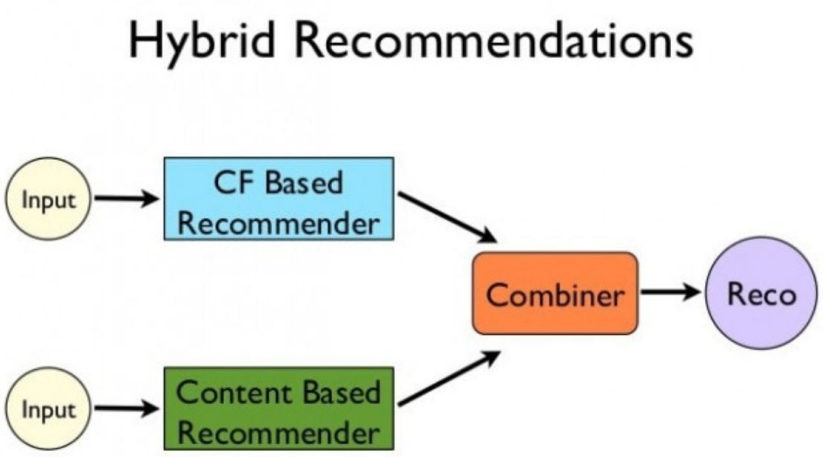

Fig 2.3 : Hybrid Recommendation systems

Several studies focused on comparing the performance of the hybrid with the pure collaborative and content-based methods and demonstrate that hybrid methods can provide more accurate recommendations than pure approaches. Such methods can be used to overcome the common problems in recommendation systems such as cold start and the data paucity problem.

Netflix is a good example of the use of hybrid recommender systems. The website makes recommendations by comparing the watching and searching habits of similar users (i.e., collaborative filtering) as well as by offering movies that share characteristics with films that a user has rated highly (content-based filtering).

CHAPTER 3: METHODOLOGIES

TF-IDF

Tf-idf stands for *term frequency-inverse document frequency*, and the tf-idf weight is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. Variations of the tf-idf weighting scheme are often used by search engines as a central tool in scoring and ranking a document's relevance given a user query.

One of the simplest ranking functions is computed by summing the tf-idf for each query term; many more sophisticated ranking functions are variants of this simple model.

Tf-idf can be successfully used for stop-words filtering in various subject fields including text summarization and classification.

How to Compute

Typically, the tf-idf weight is composed by two terms: the first computes the normalized Term Frequency (TF), aka. the number of times a word appears in a document, divided by the total number of words in that document; the second term is the Inverse Document Frequency (IDF), computed as the logarithm of the number of the documents in the corpus divided by the number of documents where the specific term appears.

- TF: Term Frequency, which measures how frequently a term occurs in a document. Since every document is different in length, it is possible that a term would appear much more times in long documents than shorter ones. Thus, the term frequency is often divided by the document length (aka. the total number of terms in the document) as a way of normalization:
- TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document).
- IDF: Inverse Document Frequency, which measures how important a term is. While computing TF, all terms are considered equally important. However it is known that certain terms, such as "is", "of", and "that", may appear a lot of times but have little importance. Thus we need to weigh down the frequent terms while scale up the rare ones, by computing the following:
- IDF(t) = log_e(Total number of documents / Number of documents with term t in it).
- Tf-Idf weight is the product of the Tf and Idf quantities.

Example

Consider a document containing 100 words wherein the word *cat* appears 3 times. The term frequency (i.e., tf) for *cat* is then (3 / 100) = 0.03. Now, assume we have 10 million documents and the word *cat* appears in one thousand of these. Then, the inverse document frequency (i.e., idf) is calculated as log(10,000,000 / 1,000) = 4. Thus, the Tf-idf weight is the product of these quantities: 0.03 * 4 = 0.12.

COSINE SIMILARITY

Well cosine similarity is a measure of similarity between two non zero vectors. One of the beautiful thing about vector representation is we can now see how closely related two sentence are based on what angles their respective vectors make.

Cosine value ranges from -1 to 1.So if two vectors make an angle 0, then cosine value would be 1, which in turn would mean that the sentences are closely related to each other.If the two vectors are orthogonal, i.e. cos 90 then it would mean that the sentences are almost unrelated.

It is evident in fig given below, the similarity between two vectors in first image is the highest i.e. ~ 1. While cosine similarity in third plot would be around -1. Cosine similarity of vectors, orthogonal to each other will be 0.

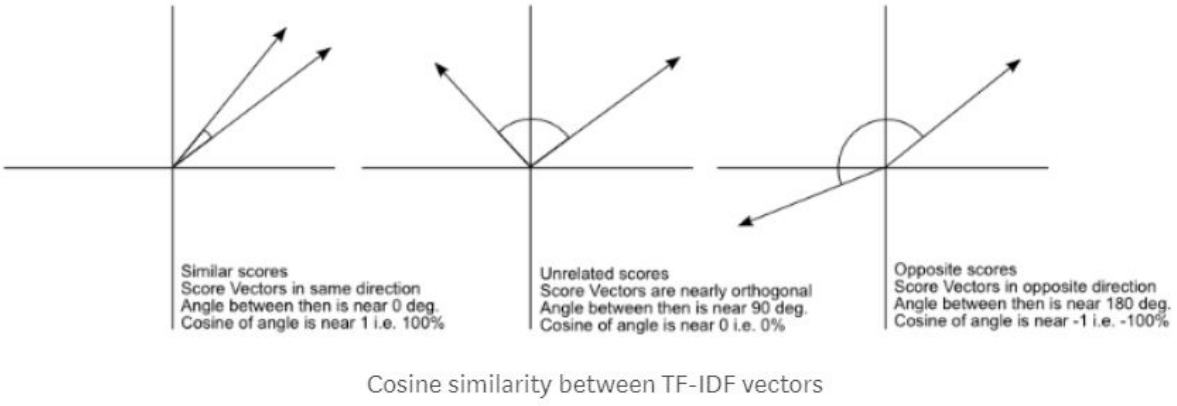

Fig 3.1 : Cosine similarity of TF-IDF vectors

COUNTVECTORIZER

The CountVectorizer provides a simple way to both tokenize a collection of text documents and build a vocabulary of known words, but also to encode new documents using that vocabulary.

You can use it as follows:

1. Create an instance of the *CountVectorizer* class.
2. Call the *fit()* function in order to learn a vocabulary from one or more documents.
3. Call the *transform()* function on one or more documents as needed to encode each as a vector.

An encoded vector is returned with a length of the entire vocabulary and an integer count for the number of times each word appeared in the document.The encoded vectors can then be used directly with a machine learning algorithm.

MATRIX FACTORIZATION

While user-based or item-based collaborative filtering methods are simple and intuitive, Matrix Factorization techniques are usually more effective because they allow us to discover the latent features underlying the interactions between users and items. We don't actually know these latent features. SVD is a matrix factorization technique that is usually used to reduce the number of features of a data set by reducing space dimensions from N to K where K < N. The matrix factorization is done on the user-item ratings matrix. From a high level, matrix factorization can be thought of as finding 2 matrices whose product is the original matrix.

Each item can be represented by a vector q_i. Similarly each user can be represented by a vector p_u such that the dot product of those 2 vectors is the expected rating.

Expected Rating $\hat{r}_{ui} = q_i^T p_u$,

Regularization

q_i and p_u can be found in such a way that the square error difference between their dot product and the known rating in the user-item matrix is minimum and p_u can be found in such a way that the square error difference between their dot product and the known rating in the user-item matrix is minimum. For the model to generalize well and not overfit the training set, a penalty

term is introduced to the minimization equation. This is represented by a regularization factor λ multiplied by the square sum of the magnitudes of user and item vectors.

$$min(p,q) \sum_{(u,i) \in K} (r_{ui} - q_i^T . p_u)^2 + \lambda (\|q_i\|^2 + \|p_u\|^2)$$

To reduce the error between the predicted and actual value, the algorithm make use of some characteristics of the dataset. In particular for each user-item *(u,i)* pair we can extract 3 parameters. μ which is the average ratings of all items, b_i which is the average rating of item i minus μ and b_u which is the average rating given by user u minus μ which makes the expected rating:

$$\widehat{r_{ui}} = q_i^T . p_u + \mu + b_i + b_u$$

And the equation to minimize becomes:

$$min(p,q,b_i,b_u) \sum_{(u,i) \in K} (r_{ui} - q_i^T . p_u - \mu - b_i - b_u)^2 + \lambda (\|q_i\|^2 + \|p_u\|^2 + b_i^2 + b_u^2)$$

CHAPTER 4: DEVELOPING JOB RECOMMENDATION SYSTEM

Datasets

Four separate Datasets were collected given as following:

1) JobsUSData.csv - This dataset consists of all the job openings in United States of America which has columns like Job Title, State where the job is present, Description, JobID, City where the job is available, start and end dates.
2) Users.tsv - This dataset consists of the User Description and all the details regarding the users. Some of the important features include UserID, City,State, Zipcode, Degree type, Total years of experience.
3) Apps.tsv - This dataset contains information about applications made by users to jobs. Each row describes an application. The UserID, WindowID, Split, and JobID columns have the same meanings as above, and the ApplicationDate column indicates the date and time at which UserID applied to JobId.

PROBLEM STATEMENT

Using the given Datasets, the main goal is to implement both content based filtering and collaborative based filtering recommender system.

In Content based filtering,The jobs are recommended using the title of a particular user, so that what other similar jobs are suitable for him.

In Collaborative based filtering, Item based recommendation are made. From the jobs which the users have applied, we get similar users and the jobs of the other similar users are recommended to other users.

IMPLEMENTATION

JOB RECOMMENDATION USING CONTENT BASED FILTERING

1) Reading the Dataset: The Dataset is read using pandas and we observe that there are 1,090,462 rows. Since the Dataset is too large and the required RAM/Memory was not sufficient, first 20,000 rows were selected as the original dataset which is used for further steps.
2) Computing TF-IDF Matrix and Similarity matrix: Using the scikit learn libraries, TF-IDF Vectorizer, linear kernel is imported. TF- IDF vector is initialised for the built in 'english' words. From the TF-IDF vectoriser, TF-IDF matrix is computed for the description column of the Jobs data. Similarity within the rows of matrices are obtained by computing the linear kernel transformation between the tf-idf matrix on itself.
3) Computing Count matrix and Cosine Similarity: Using the scikit learn libraries, Count Vectorizer and Cosine Similarity are imported. Count Vectorizer is initialised, and count matrix is obtained for the description column of the Jobs Data. Similarity within the rows of the matrices are obtained by computing the cosine_similarity between the count matrix on itself.
4) Top N Job Recommenders: A function is written which takes the job title and the count, which is the number of top jobs recommended according to the higher cosine similarity value.

JOB RECOMMENDATION USING COLLABORATIVE BASED FILTERING

1) Reading the Dataset: The same Dataset consisting of 20,000 rows is used. Along with it users.tsv consisting of first 15,000 rows and apps.tsv are read using the pandas.
2) Data Preprocessing: The Nan values, null values are filled with a null string. The three columns of Users.tsv of Majors, Years of Experience and Degree type are combined to form a single column named 'Details'.

3) Computing TF-IDF Matrix and Cosine Similarity: The TF-IDF matrix and cosine similarity are computed as described previously.
4) Top N SImilar Users: A function is written which takes the user id and the count, which is the number of top similar users according to the higher cosine similarity value. Top similar users are returned in this function.
5) Prediction of Similar Jobs from the similar ID list: The similar users obtained is stored in a list. From these similar users obtained we get the jobs which they have applied to from the JobsData and using the Job number, we recommend the new jobs to the present user. The resulting jobs is displayed as a dataframe which includes columns like JobId, Title, Description, State and City.

RESULTS

1) Count Vectorizer Method

The Fig 4.1 shows top 10 similar jobs for the given job ID using CountVectorizer method.

```
get_recommnedation('Route Delivery Drivers',10)
```

```
[3, 7517, 7601, 16340, 7167, 15251, 3776, 6420, 3992, 10566]
Route Delivery Drivers
Class A Route Drivers
Delivery Route Driver
Route Supervisor
Delivery Supervisor
CDL Class A Drivers Needed!!
Driver - Class A CDL
Experienced Delivery Driver
Class A Delivery Driver
Commercial Service Technician
```

Fig 4.1 : Output for CountVectorizer Method

TF-IDF Method

The Fig 4.2 shows top 10 similar jobs for the given job ID using TF-IDF method.

```
get_recommnedation('Route Delivery Drivers',10)
```

```
[3, 4027, 48, 12696, 13481, 12689, 26, 7517, 31, 16340]
Route Delivery Drivers
Route Delivery Driver
TRACTOR TRAILER DRIVER
Route Delivery Driver - Lanham, MD
Class A CDL Truck Driver-Distribution / Route Delivery Driver
Los Angeles - Route Sales Representative
EXPERIENCED ROOFERS
Class A Route Drivers
DRIVERS
Route Supervisor
```

Fig 4.2 : Output for TF-IDF Vectorizer Method

Collaborative Based Filtering

The first part gives the similar user Id for user ID 80. The second part gives the jobs applied by those 5 users and these new jobs are recommended to user ID 80.

```
In[43]: get_recommendations_userwise(80,5)
```

```
Out[43]: [2, 7, 9, 15, 18]
```

```
In[45]: get_job_id(get_recommendations_userwise(80,5))
```

Out[45]:

	JobID	Title	Description	State	City
478628	309823	Teller	TellerOur vision is to satisfy all our custome...	VA	Roanoke
527428	703889	Customer Service Representative, Now Accepting...	Customer Service Representative, Now Accepting...	VA	Roanoke
596359	136489	UPS Part Time Package Handler	UPS Part Time Package Handler<hr>\r\r<...	TX	Houston
657023	617374	Full Time Fork Lift Driver	Full Time Fork Lift Driver<p>The Dump is curr...	TX	Houston
680692	809208	Now Hiring All Trades	Now Hiring All TradesNow hiring all trades ASA...	TX	Houston
880059	187311	Helper	Helper<p>We are looking for: \r2 helpers ...	TX	Houston
942389	787477	Blasters	Blasters<p>We are looking for: \r2 exper...	TX	Houston
946334	828603	General Labor	General LaborGeneral Labor\r \r<br...	TX	Houston

Fig 4.3 : Output for Collaborative Filtering Method

CHAPTER 5: DEVELOPING A SONG RECOMMENDATION SYSTEM

PROBLEM STATEMENT

To Build various types of recommendation system for a song dataset and compare the performance of these recommendation system. The dataset consists of columns user_id, song_id, listen_count, title, artist, song. The user_id has a list of user id's and for each user there is a list of songs he listens to and a count of how many times he listens to that song. For each song Id it consists of the artist name and title of the song.

IMPLEMENTATION

1. **Top - N Recommendation System**

   Total listen count for the song is found by grouping the data according to its song id. The grouped data is then sorted in descending order to obtain the songs according to the popularity. Then we recommend the N trending songs. This is a very good recommendation system to tell the users which songs are currently trending. The major problem with this type of recommendation system is that it doesn't personalize the songs according to user's taste and mood.

2. **N-Similar Songs Based on Content**

   Using the scikit learn libraries, TF-IDF Vectorizer, Cosine-Similarity is imported. TF-IDF vector is initialised for the built in 'english' words. From the TF-IDF vectoriser, TF-IDF matrix is computed for the 'details' column which consists of song details like artist, song name and album name . Similarity within the rows of matrices are obtained by computing the linear kernel transformation between the tf-idf matrix on itself.

3. **N-recommendation based on Listen Count**

   User listen count was converted to user rating for this approach. This was done by grouping according to user id and finding the total listen count for that particular user.
   We then find percentage of times the user listens to each song. A rating from scale (1,5) was set for all the percentage interval. A new dataset with user Id, song Id, rating was used to feed the algorithms. A library called surprise was used to compare and build recommendation systems based on these ratings. SVD, SVD++, SlopeOne, NMF, NormalPredictor, KNNBaseline, KNNBasic, KNNWithMeans, KNNWithZScore, BaselineOnly, CoClustering algorithms were tested using 3-fold Cross Validation With 20000 data points. SVD++ algorithm was the one with lowest cross validation error.

But the time taken for building the model is significantly higher than SVD. SVD++ is computationally expensive and have slightly better accuracy than SVD. Hence SVD was used to train the model with full training dataset. Root mean squared error was obtained for the predicted results. Finally N songs were recommended for the users in test set.

RESULTS

1. Top - N Recommendation System

Top Ten songs recommended based on popularity are shown in figure 5.1.

```
TopNSongsPopularityWise(songData,10)
```

```
Top 10 songs based on Popularity are:
Sehr kosmisch
Undo
You\'re The One
Dog Days Are Over (Radio Edit)
Revelry
Horn Concerto No. 4 in E flat K495: II. Romance (Andante cantabile)
Secrets
Tive Sim
Fireflies
Hey_ Soul Sister
```

Fig.5.1 Top 10 Popular Songs

2. N-Similar Songs Based on Content

Fig.5.2.1 shows 10 songs have been recommended which has similar album title and artist.

```
TopNSongsContentBased(songData,'Stronger',10)
```

```
Top 10 Recommedations Based on Content for the Song Stronger
Champion
Homecoming
RoboCop
Celebration
Late
Say You Will
Graduation Day
Bad News
The Glory
Flashing Lights
```

Fig.5.2.1 Top 10 Recommendations for song "Stronger"

```
TopNSongsContentBased(songData,'Come To Me',10)
```

```
Top 10 Recommedations Based on Content for the Song Come To Me
Undo
Isobel
Unravel
Army of Me
Cover Me
Come Back to Me
Hidden Place
Come & Talk To Me
Possibly Maybe
Human Behaviour
```

Fig.5.2.2 : Top 10 Recommendations for song "Come To Me"

3. N -recommendation Based on Listen Count

The cross-validation error is used to find the best algorithm. Fig. 5.3.1 shows validation error for several algorithms. The SVD++ algorithm has the least error of 1.3854 and takes 0.28 seconds to compute. But SVD takes 0.05 seconds for slightly lesser accuracy. So we take SVD algorithm to train it with full training set. The root mean squared error(RMSE) for test set is 1.3536. Fig. 5.3.2 Shows top 4 recommendation for each user.

Algorithm	test_rmse	fit_time	test_time
SVDpp	1.385432	6.141294	0.282330
SVD	1.388503	0.909744	0.056076
BaselineOnly	1.417037	0.049390	0.069337
KNNWithZScore	1.516613	0.134548	0.526326
KNNWithMeans	1.523686	0.083327	0.430733
KNNBaseline	1.550297	0.111034	0.497491
SlopeOne	1.561242	1.004845	0.205155
CoClustering	1.566119	1.489116	0.076104
NMF	1.628347	2.464847	0.085745
KNNBasic	1.754106	0.063472	0.395838
NormalPredictor	2.077872	0.038141	0.079834

Fig.5.3.1 Cross-Validation Error for various algorithm

```
---------------USER ID-----------------------------------------------------SONGS-------------------------
35fd95ee6b6db837ececbadf4c94e558150d05d7 ['God', 'Only Happy When It Rains', 'Si Una Vez', 'Stupid Girl']
0819445f772b8026aeafed5f3437b38af6f9b5ff ['Cosmic Love', 'Bodies', 'Che Sara', 'Secrets']
76908d9669ee230cb63cc6fcebb1f0e077a391c1 ['You Get What You Give', 'I Gotta Feeling', 'Te lo agradezco_ pero no', 'Clocks']
b7c8a0ad9c63610e5fdf076c6163b6b3f306d7a0 ["It\\'s My Own Fault", 'A Thousand Miles', 'Gente Que No', 'Emergency (Album Versio
n)']
ff01bed0c6dfdcfb06f701c7402d7c51dde165e6 ['Scream', 'Robot Soul (Radio Edit)', 'I Gotta Feeling']
81cb90bc3b2763d98a62e2bea6f46d1aac5aa8e2 ['Undo', 'Dog Days Are Over (Radio Edit)', 'Better To Reign In Hell', 'Halo']
600ca38d0b3c0e1a43907b49fa369b00855f936f ['Eye Of The Tiger', 'Dirty Little Secret', 'Alejandro', 'I Gotta Feeling']
99ebb0fe44b4bb3f05e264eeece1240c397e082f ['Victoria (LP Version)', "Pimpin\\'", 'Bloodstream', 'Meteor Shower']
9e15e00c217cadee8366535bab5d237d1b033212 ['Everything', 'Feelings Show', 'Home (Album Version)', 'Tides']
bc4c2cd722d98458faecdfd700ae3c0e09856ec8 ['Catch You Baby (Steve Pitron & Max Sanna Radio Edit)', 'Hey_ Soul Sister', 'Not Re
ady To Make Nice', 'Me and Your Cigarettes']
fc1a49d60fd9f968d03e8be4e0a469e036894c8c ['Bodies', "Don\\'t Worry Be Happy", 'Dancing With Tears In My Eyes', 'Vanilla Twili
ght']
3b18752736cd5ec3790b162245d83752c3a31e3b ['Invalid', 'Blister In The Sun', 'Uptown', "Day \\'N\\' Nite"]
b654849234292646657b84d64ce5871f4b1199da ['Paper Aeroplane', 'Babylon', "We\\'re Going To Be Friends", 'A Book Like This']
986e4baf17f079f3772cbb4030201cbb17a61534 ['Kid', 'Black River', 'Bottom Of the Barrel', "Dreamin\\'"]
5c74627c585a5574f8b01751cdd4a436c03ec4a7 ['Sayonara-Nostalgia', 'Revelry', 'Peces', 'Two Is Better Than One']
```

Fig.5.3.2 Top 4 Songs for users in test set

CHAPTER 6 : REFERENCES

[1] G. Adomavicius and A. Tuzhilin, “Towards the next generation of recommender systems: a survey of the state-of-the-art and possible extensions,” *IEEE Trans. on Data and Knowledge Engineering* 17:6, pp. 734– 749, 2005.

[2] Sugiyama K. and Kan M.-Y., "A comprehensive evaluation of scholarly paper recommendation using potential citation papers," *International Journal on Digital Libraries,* vol. 16, pp. 91–109, 2015.

APPENDIX A - Python Code For Job Recommendation System

1) Content Based Filtering

Importing all the libraries and Dataset

```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python docker image: https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load in

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the "../input/" directory.
# For example, running this (by clicking run or pressing Shift+Enter) will list the files in the input directory

import os
print(os.listdir("../input"))

# Any results you write to the current directory are saved as output.

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import linear_kernel
```

```
jobsUSData = pd.read_csv('../input/000000/jobsUsdata.csv')
jobsUSData.head()
```

```
jobsUSbase = jobsUSData.iloc[0:20000]
jobsUSbase
```

Computing TF-IDF Matrix and Cosine Similarity

```
tf = TfidfVectorizer(analyzer='word',ngram_range=(1,2),min_df=0,stop_words='english')
tfidf_matrix = tf.fit_transform(jobsUSbase['Description'])
tfidf_matrix.shape
```

```
(20000, 883828)
```

```
cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)
cosine_sim[0]
```

```
array([1.        , 0.03571031, 0.00526099, ..., 0.04434704, 0.00279839,
       0.04338058])
```

Computing Count Matrix and Cosine Similarity

```
count = CountVectorizer(stop_words='english')
count_matrix = count.fit_transform(jobsUSbase['Description'])

count_matrix.shape

cosine_sim2 = cosine_similarity(count_matrix, count_matrix)
```

Taking the titles and resetting the index

```
jobsUSbase.reset_index()

titles  = jobsUSbase['Title']
indices = pd.Series(jobsUSbase.index, index=jobsUSbase['Title'])
```

Function for Top N Job recommenders

```
def get_recommnedation(title,cnt):
    similar_job_indices = []
    #def get_recommendation(title, no_of_jobs):
    i = indices[title]
    simulated_scores = enumerate(cosine_sim[i])
    simulated_scores = sorted(simulated_scores, key = lambda x: x[1],reverse = True)
    #print(simulated_scores)
    count = 0
    for i in simulated_scores:
        count = count+1
        similar_job_indices.append(i[0])
        if count == cnt:
            break
    print(similar_job_indices)
    for i in similar_job_indices:
        print(titles[i])

get_recommnedation('Route Delivery Drivers',10)
```

2) Collaborative Filtering

Reading the required Datasets and Data Preprocessing

```
userData = pd.read_csv('../input/000002/users.tsv',delimiter = '\t')
user_based_approach_US = userData.loc[userData['Country']=='US']
user_based_approach_US = user_based_approach_US.iloc[0:15000]
appsData = pd.read_csv('../input/000003/apps.tsv',delimiter = '\t')

user_based_approach_US['Major'] = user_based_approach_US['Major'].fillna('')
user_based_approach_US['DegreeType'] = user_based_approach_US['DegreeType'].fillna('')
user_based_approach_US['TotalYearsExperience'] = str(user_based_approach_US['TotalYearsExperience'].fillna(''))
user_based_approach_US['Details'] = user_based_approach_US['Major'] + user_based_approach_US['DegreeType'] + user_based_approach_US['TotalYea
```

Computing TF-IDF Matrix and Cosine Similarity

```
tf = TfidfVectorizer(analyzer = 'word', ngram_range = (1,2), min_df =0, stop_words = 'english')
tfidf_matrix = tf.fit_transform(user_based_approach_US['Details'])
tfidf_matrix.shape
```

```
(15000, 8419)
```

```
cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)
cosine_sim[0]
```

```
array([1.        , 0.72866229, 0.91832588, ..., 0.91832588, 0.70380466,
       0.83574527])
```

Taking the User IDs and Resetting the Index

```
user_based_approach_US = user_based_approach_US.reset_index()
userid = user_based_approach_US['UserID']
indices = pd.Series(user_based_approach_US.index, index=user_based_approach_US['UserID'])
```

Function for returning Top N Similar users

```
def get_recommendations_userwise(userid,cnt):
    user_indices = []
    idx = indices[userid]
    sim_scores = enumerate(cosine_sim[idx])
    sim_scores = sorted(sim_scores, key = lambda x:x[1], reverse = True)
    count = 0
    for i in sim_scores:
        count = count+1
        user_indices.append(i[0])
        if count == cnt:
            break
    return user_indices

print("---- Top N similar users with userId: 123 ---")
get_recommendations_userwise(123,10)
```

```
---- Top N similar users with userId: 123 ---
```

```
[4, 165, 246, 1152, 1162, 1460, 1617, 1815, 2128, 2173]
```

Function for returning similar Jobs from similar users

```
def get_job_id(userid_list):
    jobs_userwise = appsData['UserID'].isin(userid_list)
    df1 = pd.DataFrame(data = appsData[jobs_userwise], columns=['JobID'])
    joblist = df1['JobID'].tolist()
    Job_list = jobsUSData['JobID'].isin(joblist)
    df_temp = pd.DataFrame(data = jobsUSData[Job_list], columns = ['JobID','Title','Description','State','City'])
    return df_temp
get_recommendations_userwise(80,5)
get_job_id(get_recommendations_userwise(80,5))
```

	JobID	Title	Description	State	City
478628	309823	Teller	TellerOur vision is to satisfy all our custome...	VA	Roanoke
527428	703889	Customer Service Representative, Now Accepting...	Customer Service Representative, Now Accepting...	VA	Roanoke
596359	136489	UPS Part Time Package Handler	UPS Part Time Package Handler<hr>\r\r<...	TX	Houston
657023	617374	Full Time Fork Lift Driver	Full Time Fork Lift Driver<p>The Dump is curr...	TX	Houston
680692	809208	Now Hiring All Trades	Now Hiring All TradesNow hiring all trades ASA...	TX	Houston
880059	187311	Helper	Helper<p>We are looking for: \r2 helpers ...	TX	Houston
942389	787477	Blasters	Blasters<p>We are looking for: \r2 exper...	TX	Houston
946334	828603	General Labor	General LaborGeneral Labor\r \r<br...	TX	Houston

APPENDIX B - Python Code For Song Recommendation System

Importing all Libraries

```
import numpy as np
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import linear_kernel,cosine_similarity
from surprise import Dataset, evaluate,Reader,SVD,SVDpp
from surprise import SlopeOne, NMF, NormalPredictor, KNNBaseline, KNNBasic
from surprise import KNNWithMeans, KNNWithZScore, BaselineOnly, CoClustering
from surprise.model_selection import cross_validate
from surprise import KNNBasic
from surprise.model_selection import train_test_split
from surprise import accuracy
from collections import defaultdict
```

Reading the Dataset

```
songData = pd.read_csv('song_data.csv')
songData.head(4)
```

Code for Top-N Recommendation System

```
def TopNSongsPopularityWise(songdata,Count):
    dfSongId = songdata.groupby(['song_id','title']).agg({'listen_count': 'count'})
    dfSongId = dfSongId.sort_values('listen_count', ascending = False).reset_index()
    topSongs = list(dfSongId['title'])
    print('Top {} songs based on Popularity are:'.format(Count))
    for i in range(0,Count):
        print(topSongs[i])
TopNSongsPopularityWise(songData,10)
```

Code N-Similar Songs Recommendation System

```
def TopNSongsContentBased(songdata,title,Count):
    songdata.drop_duplicates(subset = 'title',keep ='first',inplace = True)
    songdata.reset_index(drop=True, inplace=True)
    songdata['Details'] = songdata['artist'] + songdata['title'] + songdata['song']
    tf = TfidfVectorizer(analyzer='word',ngram_range=(1,2),min_df=0,stop_words='english')
    tfidf_matrix = tf.fit_transform(songdata['Details'])
    cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)
    titles  = songdata['title']
    indices = pd.Series(songdata.index, index=songdata['title'])
    def get_recommnedation(title,cnt):
        similarSongIndices = []
        i = indices[title]
        simulatedScores = enumerate(cosine_sim[i])
        simulatedScores = sorted(simulatedScores, key = lambda x: x[1],reverse = True)
        #print(simulated_scores)
        count = -1
        for i in simulatedScores:
            count = count+1
```

```
            similarSongIndices.append(i[0])
            if count == cnt:
                break
        return similarSongIndices
    similarSongIndices = get_recommnedation(title,Count)
    print('Top',Count,'Recommedations Based on Content for the Song',title)
    for i in range(1,len(similarSongIndices)):
        print(titles[similarSongIndices[i]])
TopNSongsContentBased(songData,'Come To Me',10)
```

```
listenCountAvgDf = songData.groupby('user_id').agg({'listen_count': 'count'})
listenCountAvgDf = listenCountAvgDf[listenCountAvgDf['listen_count']>10]
UsersSongDataDf = pd.merge(songData, listenCountAvgDf, how='inner', on=['user_id'])
UsersSongDataDf = UsersSongDataDf.dropna()
```

Function to compute ratings explicitly

```
#Finding the rating given by the user Explicitly by Taking the percentage
def SongRating(x,y):
    res = (x/y)*100
    if(res <=4):
        return 1
    elif (4<res<=6):
        return 2
    elif(6<res<=8):
        return 3
    elif(8<res<=11):
        return 4
    elif(11<res):
        return 5

UsersSongDataDf['UserSongRating'] = UsersSongDataDf.apply(lambda x: SongRating(x['listen_count_x'],x['listen_count_y']), axis=1)
```

Code to perform cross validation for all the algorithms

```
UsersSongDatadf = UsersSongDataDf[0:20000]
reader = Reader(rating_scale=(1, 5))
data = Dataset.load_from_df(UsersSongDatadf[['user_id', 'title', 'UserSongRating']], reader)
benchmark = []
algorithms = [SVD(), SVDpp(), SlopeOne(), NMF(), NormalPredictor(), KNNBaseline(),
              KNNBasic(), KNNWithMeans(), KNNWithZScore(), BaselineOnly(), CoClustering()]
# Iterate over all algorithms
for algorithm in algorithms:
    # Perform cross validation
    results = cross_validate(algorithm, data, measures=['RMSE'], cv=3, verbose=False)

    # Get results & append algorithm name
    tmp = pd.DataFrame.from_dict(results).mean(axis=0)
    tmp = tmp.append(pd.Series([str(algorithm).split(' ')[0].split('.')[-1]], index=['Algorithm']))
    benchmark.append(tmp)

pd.DataFrame(benchmark).set_index('Algorithm').sort_values('test_rmse')
```

Building the model with SVD algorithm

```
data = Dataset.load_from_df(UsersSongDataDf[['user_id', 'title', 'UserSongRating']], reader)
trainset, testset = train_test_split(data, test_size=0.2)
algo = SVD()

# Train the algorithm on the trainset, and predict ratings for the testset
algo.fit(trainset)
predictions = algo.test(testset)

# Then compute RMSE
accuracy.rmse(predictions)
```

Printing the Recommended songs to users

```
def get_top_n(predictions, n=10):

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

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

    return top_n
top_n = get_top_n(predictions, n=4)
print('---------------USER ID-----------------------------------------------------SONGS-------------------------')
for uid, user_ratings in top_n.items():
    print(uid, [iid for (iid, _) in user_ratings])
```