International Institute of Information Technology, Hyderabad

Semester Project Report (Spring 2015, PG)

Faculty - ReddyRaja A

Machine Learning with Spark

– A Movie/Program recommendation

Submitted by,

Nageswara Pavan Kumar S D

(Roll number – 201150836)

1 INTRODUCTION **4**

2. MACHINE LEARNING SYSTEMS **5**

2.1 MACHINE LEARNING MODELS 6
2.2 COMPONENTS OF A MACHINE LEARNING SYSTEM 6
2.2.1 Data ingestion and storage *7*
2.2.2 Data cleansing and transformation *7*
2.2.3 Model training and testing loop *9*
2.2.4 Model deployment and integration *9*

3 RECOMMENDATION SYSTEMS **10**

3.1 COLLABORATIVE FILTERING 11
3.1.1 User based Collaborative Filtering *12*
3.1.1.1 Similarity Measures 12
3.1.2 Item based Collaborative Filtering *14*
3.1.2.1 Similarity Measures 15
3.2 MATRIX FACTORIZATION 15
3.2.1 Implicit matrix factorization *18*
3.2.2 Alternating least squares *19*
3.3 CONTENT-BASED FILTERING 20
3.4 HYBRID RECOMMENDER SYSTEMS 21

4 PARALLEL PROGRAMMING WITH APACHE SPARK **23**

4.1 INTRODUCTION TO CORE SPARK CONCEPTS 24
4.1.1 Spark Clusters *24*
4.1.2 Programming model *25*
4.1.3 Lazy Evaluation *27*
4.2 MACHINE LEARNING WITH MLLIB 28
4.2.1 Statistics *29*
4.2.2 COLLABORATIVE FILTERING AND RECOMMENDATION 30
4.2.2.1 Alternating Least Squares 30

5 RECOMMENDATION SYSTEM WITH APACHE SPARK **32**

5.1 EXPLORING AND PREPARING DATA 32
5.2 ALS BASED RECOMMENDATION SYSTEM 37
5.2.1 Setup and execution *38*
5.2.2 Training using ALS *39*
5.2.3 Training a model using implicit feedback data *40*
5.2.4 User based prediction *40*

5.2.5 Item based prediction *41*
5.3 MODEL EVALUATION AND RESULTS 42
5.3.1 Mean Squared Error *42*

6 CONCLUSION 45

APPENDIX A 46

APPENDIX B 48

APPENDIX C 55

APPENDIX D 57

REFERENCES: 58

1 Introduction

The advent of Internet and rapid growth of applications makes now available to the users a large amount of information, products and services. As a result many modern applications expose to users a huge collection of items to choose from. In such scenarios recommendation techniques are often presented as a solution to the information overload problem by helping the users to filter relevant items on the basis of their needs and preferences. Such systems typically provide the user with a list of recommended items they might prefer, or predict how much they might prefer each item. These systems help users to decide on appropriate items, and ease the task of finding preferred items in the collection.

For instance, with the recent expansion of TV content, digital TV networks and broadband, smarter TV entertainment is needed as well. As there are several hundreds of available programs every day, users need a means to easily find the interesting ones and watch such programs at that preferred time of day.

This report aims to present a study on how machine learning techniques can be applied to such a requirement to recommend TV programs via distributed learning system that uses Spark as its computation engine. A brief description of machine learning models, their relevance and their components is discussed. These techniques are connected with various paradigms of Recommendation methods considering in detail the mathematical modelling for ALS formulation in the context of TV programs. Also a brief outline of Apache Spark is included to introduce the higher level API within the machine learning library.

The rest of the report aims to focus on building a recommendation engine based on Apache Spark. This includes various steps such as data analysis and identifying relevant features, implementing a suitable recommendation model, training and evaluation of recommendations.

2. Machine learning systems

Machine learning systems aim at transforming of data of various magnitudes into actionable form. As the amount of data grows, the size and complexity of datasets grows and it becomes difficult for human involvement to uncover various patterns. A model driven and statistical approach such as machine learning can be very effective to handle these datasets and lift the burden of time-consuming tasks.

Considering the cases of personalized recommendations and predictive analysis, machine learning systems are proven to be highly beneficial. Personalization being of the most potential applications refers to adapting the experience of a user and the content presented to them based on various factors, which might include user behavior data as well as external factors. While generally recommendations and personalization are focused on a one-to-one situation, segmentation approaches might try to assign users into groups based on characteristics and, possibly, behavioral data. The approach might be fairly simple or might involve a machine learning model such as clustering. Either way, the result is a set of segment assignments that might allow us to understand the broad characteristics of each group of users, what makes them similar to each other within a group, and what makes them different from others in different groups.

Predictive modelling and analytics is another area where machine learning techniques can be effective. This encompasses recommendations, personalization and targeting too. the term predictive modeling to refer to other models that seek to make predictions. An example of is can be a model to predict the potential viewing activity and thus revenue of new TV program before any data is available on how popular the show could be. Based on past activity and revenue data, together with content attributes, a regression model is created and this that can be used to make predictions for brand new titles.

2.1 Machine learning models

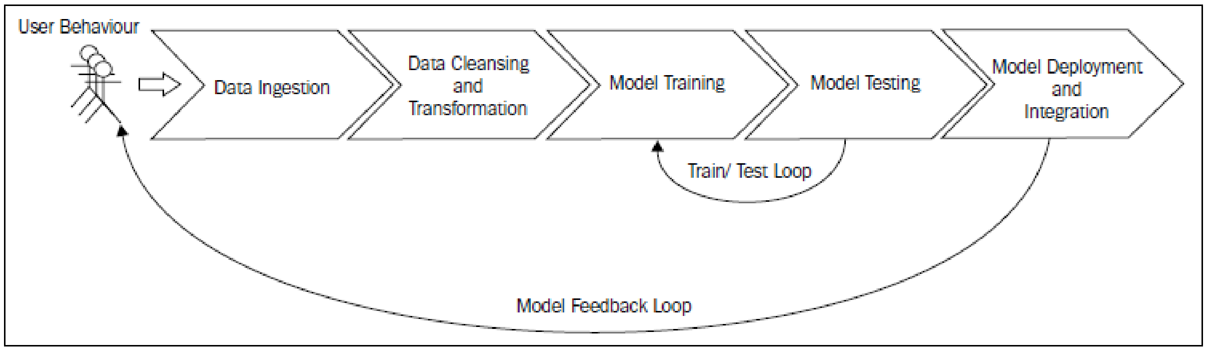
The scenarios for application of machine learning techniques can be broadly divided into two categories.

Supervised learning: These types of models use labeled data to learn. Recommendation engines, regression, and classification are examples of supervised learning methods. The labels in these models can be user-movie ratings (for recommendation), movie tags (in the case of the classification), or revenue figures (for regression).

Unsupervised learning: When a model does not require labeled data, we refer to unsupervised learning. These types of models try to learn or extract some underlying structure in the data or reduce the data down to its most important features. Clustering, dimensionality reduction, and some forms of feature extraction, such as text processing, are all unsupervised techniques

2.2 Components of a machine learning system

The high level components of a data driven machine learning system can be outlined as below



A general machine learning pipeline

2.2.1 Data ingestion and storage

The first step in a machine learning pipeline will be taking in the data that is required for training models. In general, data is generated by user activity; this is augmented by the data generated by supporting systems and external sources. This data can be ingested in various ways, for example, gathering user activity data from browser and application event logs. Once the collection mechanisms are in place, the data usually needs to be stored. This includes the raw data, data resulting from intermediate processing, and final model results.

2.2.2 Data cleansing and transformation

The majority of machine learning models operate on features, which are typically numerical representations of the input variables for the model. The data collected via various systems and sources in the ingestion step is, in most cases, in a raw form. For example, we might log user events such as details of when a user views the information page for a movie or a program, when they watch a movie, or when they provide some other feedback. These event logs will typically contain some combination of textual and numeric information about the event (and also, perhaps, other forms of data such as images or audio).

In order to use this raw data as part of machine learning models, in almost all cases, we need to perform preprocessing, which might include:

• Filtering data: Let’s assume that we want to create a model from a subset of the raw data, such as only the most recent few months of activity data or onlyevents that match certain criteria.

• Dealing with missing, incomplete, or corrupted data: Many real-world datasets are incomplete in some way. This might include data that is missing (for example, due to a missing user input) or data that is incorrect or flawed (for example, due to an error in data ingestion or storage, technical issues or bugs, or software or hardware failure). We might need to filter out bad data or alternatively decide a method to fill in missing data points (such as using the average value from the dataset for missing points, for example).

• Dealing with potential anomalies, errors, and outliers: Erroneous or outlier data might skew the results of model training, so we might wish to filter these cases out or use techniques that are able to deal with outliers.

• Aggregating data: Certain models might require input data that is aggregated in some way, such as computing the sum of a number of different event types per user.

Once we have performed initial preprocessing on our data, we often need to transform the data into a representation that is suitable for machine learning models. For many model types, this representation will take the form of a vector or matrix structure that contains numerical data. Common challenges during data transformation and feature extraction include:

• Taking categorical data (such as country for geolocation or category for a movie) and encoding it in a numerical representation.

• Extracting useful features from text data.

• We often convert numerical data into categorical data to reduce the number of values a variable can take on. An example of this is converting a variable for age into buckets (such as 25-35, 45-55, and so on).

• Transforming numerical features; for example, applying a log transformation to a numerical variable can help deal with variables that take on a very large range of values.

• Normalizing and standardizing numerical features ensures that all the different input variables for a model have a consistent scale. Many machine learning models require standardized input to work properly.

• Feature engineering is the process of combining or transforming the existing variables to create new features. For example, we can create a new variable that is the average of some other data, such as the average number of times a user watches a movie.

2.2.3 Model training and testing loop

Once we have the training data in a form that is suitable for our model, we can proceed with the model's training and testing phase. During this phase, we are primarily concerned with model selection. This can refer to choosing the best modeling approach for our task, or the best parameter settings for a given model. In fact, the term model selection often refers to both of these processes, as, in many cases, we might wish to try out various models and select the best performing model (with the best performing parameter settings for each model). It is also common to explore the application of combinations of different models (known as ensemble methods) in this phase.

This is typically a fairly straightforward process of running our chosen model on our training dataset and testing its performance on a test dataset (that is, a set of data that is held out for the evaluation of the model that the model has not seen in the training phase). This process is referred to as cross-validation.

However, due to the large scale of data we are typically working with, it is often useful to carry out this initial train-test loop on a smaller representative sample of our full dataset or perform model selection using parallel methods where possible. For this part of the pipeline, Spark's built-in machine learning library, MLlib, is a perfect fit.

2.2.4 Model deployment and integration

Once we have found the optimal model based on the train-test loop, we might still face the task of deploying the model to a production system so that it can be used to make actionable predictions.

Usually, this process involves exporting the trained model to a central data store from where the production-serving system can obtain the latest version. Thus, the live system refreshes the model periodically as a new model is trained.

3 Recommendation systems

Recommendations are essentially a subset of personalization. Recommendation generally refers to presenting a user with a list of items that we hope the user will be interested in. Recommendations might be used in web pages (for example, recommending related products), via e-mails or other direct marketing channels, via mobile apps, and so on. Personalization is very similar to recommendations, but while recommendations are usually focused on an explicit presentation of products or content to the user, personalization is more generic and, often, more implicit.

For example, the DVD rental provider Netflix1 displays predicted ratings for every displayed movie in order to help the user decide which movie to rent. The online book retailer Amazon2 provides average user ratings for displayed books, and a list of other books that are bought by users who buy a specific book. Microsoft provides many free down-loads for users, such as bug fixes, products and so forth. When a user downloads some software, the system presents a list of additional items that are downloaded together. All these systems are typically categorized as recommender systems, even though they provide diverse services. So with a perspective of Recommender system as a function,

Given: User model (e.g. ratings, preferences, demographics, situational context) and Items (with or without description of item characteristics)

Find: Relevance score. Used for ranking

Finally: Recommend items that are assumed to be relevant

But: Relevance might be context-dependent, Characteristics of the list itself might be important (diversity)

Recommender systems address information overload problems by applying artificial intelligence techniques such as user modeling, content-based and collaborative filtering, case-based reasoning, etc.

3.1 Collaborative filtering

One approach to the design of recommender systems that has seen wide use is collaborative filtering. Collaborative filtering methods are based on collecting and analyzing a large amount of information on users' behaviors, activities or preferences and predicting what users will like based on their similarity to other users. A key advantage of the collaborative filtering approach is that it does not rely on machine analyzable content and therefore it is capable of accurately recommending complex items such as movies without requiring an "understanding" of the item itself. Many algorithms have been used in measuring user similarity or item similarity in recommender systems.

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. When building a model from a user's profile, a distinction is often made between explicit and implicit forms of data collection.

Examples of explicit data collection include the following actions:

- Asking a user to rate an item on a sliding scale.
- Asking a user to search.
- Asking a user to rank a collection of items from favorite to least favorite.
- Presenting two items to a user and asking him/her to choose the better one of them.
- Asking a user to create a list of items that he/she likes.

Examples of implicit data collection include the following:

- Observing the items that a user views in an online store.
- Analyzing item/user viewing times.
- Keeping a record of the items that a user purchases online.
- Obtaining a list of items that a user has listened to or watched on his/her computer.
- Analyzing the user's social network and discovering similar likes and dislikes

The two primary areas of collaborative filtering are the Neighborhood Models and Matrix Factorization Models. Neighborhood methods are centered on computing the relationships between items or, alternatively, between users.

3.1.1 User based Collaborative Filtering

In a user-based approach, if two users have exhibited similar preferences (that is, patterns of interacting with the same items in broadly the same way), then we would assume that they are similar to each other in terms of taste. To generate recommendations for unknown items for a given user, we can use the known preferences of other users that exhibit similar behavior. We can do this by selecting a set of similar users and computing some form of combined score based on the items they have shown a preference for. The overall logic is that if others have tastes similar to a set of items, these items would tend to be good candidates for recommendation.

Computing Predictions: Besides the rating matrix R, a user–user CF system requires a similarity function s:U×U → R computing the similarity between two users and a method for using similarities and ratings to generate predictions.

To generate predictions or recommendations for a user u, user–user CF first uses s to compute a neighborhood N ⊆U of neighbors of u.

Once N has been computed, the system combines the ratings of users in N to generate predictions for user u’s preference for an item i. This is typically done by computing the weighted average of the neighboring users’ ratings i using similarity as the weights:

$$p_{u,i} = \bar{r}_u + \frac{\sum_{u' \in N} s(u,u')(r_{u',i} - \bar{r}_{u'})}{\sum_{u' \in N} |s(u,u')|}$$

3.1.1.1 Similarity Measures

A critical design decision in implementing user–user CF is the choice of similarity function. Several different similarity functions have been proposed and evaluated in the literature.

Pearson correlation: This method computes the statistical correlation

(Pearson's r) between two user's common ratings to determine their similarity. GroupLens and BellCore both used this method. The correlation is computed by the following:

$$s(u,v) = \frac{\sum_{i \in I_u \cap I_v} (r_{u,i} - \bar{r}_u)(r_{v,i} - \bar{r}_v)}{\sqrt{\sum_{i \in I_u \cap I_v} (r_{u,i} - \bar{r}_u)^2} \sqrt{\sum_{i \in I_u \cap I_v} (r_{v,i} - \bar{r}_v)^2}}$$

Pearson correlation suffers from computing high similarity between users with few ratings in common. This can be alleviated by setting a threshold on the number of co-rated items necessary for full agreement (correlation of 1) and scaling the similarity when the number of co-rated items falls below this threshold. Experiments have shown a threshold value of 50 to be useful in improving prediction accuracy, and the threshold can be applied by multiplying the similarity function by min{|Iu∩Iv|/50}.

Spearman rank correlation: The Spearman rank correlation coefficient is another candidate for a similarity function. For the Spearman correlation, the items a user has rated are ranked such that their highest-rated item is at rank 1 and lowerrated items have higher ranks. Items with the same rating are assigned the average rank for their position. The computation is then the same as that of the Pearson correlation, except that ranks are used in place of ratings.

Cosine similarity: This model is somewhat different than the previously described approaches, as it is a vector-space approach based on linear algebra rather than a statistical approach. Users are represented as |I|-dimensional vectors and similarity is measured by the cosine distance between two rating vectors. This can be computed efficiently by taking their dot product and dividing it by the product of their L2 (Euclidean) norms:

$$s(u,v) = \frac{\mathbf{r}_u \cdot \mathbf{r}_v}{\|\mathbf{r}_u\|_2 \|\mathbf{r}_v\|_2} = \frac{\sum_i r_{u,i} r_{v,i}}{\sqrt{\sum_i r_{u,i}^2}\sqrt{\sum_i r_{v,i}^2}}$$

Unknown ratings are considered to be 0; this causes them to effectively drop out of the numerator. If the user mean baseline is subtracted from the ratings prior to computing the similarity, cosine similarity is equivalent to Pearson correlation when the users have rated the same set of items.

3.1.2 Item based Collaborative Filtering

We can also take an item-based approach that computes some measure of similarity between items. This is usually based on the existing user-item preferences or ratings. Items that tend to be rated the same by similar users will be classed as similar under this approach. Once we have these similarities, we can represent a user in terms of the items they have interacted with and find items that are similar to these known items, which we can then recommend to the user. Again, a set of items similar to the known items is used to generate a combined score to estimate for an unknown item.

In real-valued ratings domains, the similarity scores can be used to generate predictions using a weighted average, similar to the procedure used in user–user CF. Recommendations are then generated by picking the candidate items with the highest predictions.

After collecting a set S of items similar to i, pu,i can be predicted as follows:

$$p_{u,i} = \frac{\sum_{j \in S} s(i,j) r_{u,j}}{\sum_{j \in S} |s(i,j)|}$$

S is typically selected to be the k items most similar to j that u has also rated for some neighborhood size k

3.1.2.1 Similarity Measures

The item–item prediction process requires an item–item similarity matrix S. This matrix is a standard sparse matrix, with missing values being 0 (no similarity); it differs in this respect from R, where missing values are unknown.

As in user–user collaborative filtering, there are a variety of methods that can be used for computing item similarities.

Cosine similarity. Cosine similarity between item rating vectors is the most popular similarity metric, as it is simple, fast, and produces good predictive accuracy.

$$s(i,j) = \frac{\mathbf{r}_i \cdot \mathbf{r}_j}{\|\mathbf{r}_i\|_2 \|\mathbf{r}_j\|_2}$$

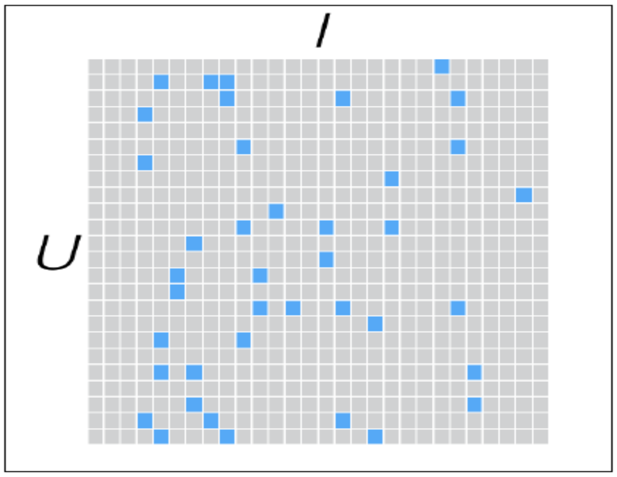
Pearson correlation. Pearson correlation has also been proposed for item–item recommendation, but does not seem to work as well as cosine similarity.

The user- and item-based approaches are usually referred to as nearest-neighbor models, since the estimated scores are computed based on the set of most similar users or items (that is, their neighbors).

3.2 Matrix factorization

Some of the most successful realizations of latent factor models are based on matrix factorization. In its basic form, matrix factorization characterizes both items and users by vectors of factors inferred from item rating patterns. High correspondence between item and user factors leads to a recommendation. These methods have become popular in recent years by combining good scalability with predictive accuracy. In addition, they offer much flexibility for modeling various real-life situations.

Recommender systems rely on different types of input data, which are often placed in a matrix with one dimension representing users and the other dimension representing items of interest. The most convenient data is highquality explicit feedback, which includes explicit input by users regarding their interest in products. We refer to explicit user feedback as ratings. Usually, explicit feedback comprises a sparse matrix, since any single user is likely to have rated only a small percentage of possible items.

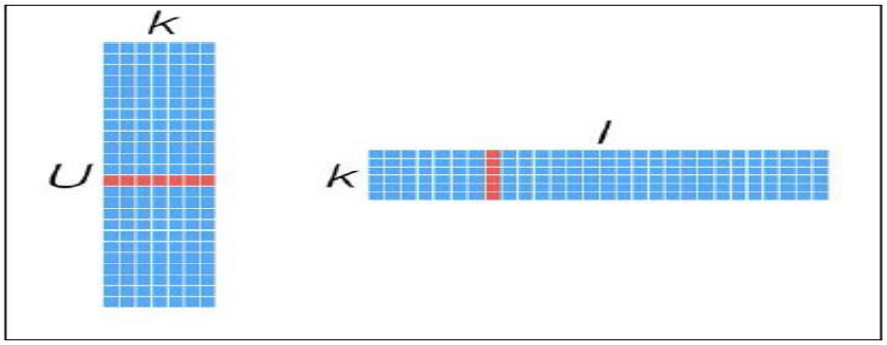


One strength of matrix factorization is that it allows incorporation of additional information. When explicit feedback is not available, recommender systems can infer user preferences using implicit feedback, which indirectly reflects opinion by observing user behavior including purchase history, browsing history, search patterns, or even mouse movements. Implicit feedback usually denotes the presence or absence of an event, so it is typically represented by a densely filled matrix.

Matrix factorization models map both users and items to a joint latent factor pace of dimensionality f, such that user-item interactions are modeled as inner products in that space.

Accordingly, each item i is associated with a vector qi ∈・ f, and each user u is associated with a vector pu ∈ ・ f. For a given item i, the elements of qi measure the extent to which the item possesses those factors, positive or negative.

For a given user u, the elements of pu measure the extent of interest the user has in items that are high on the corresponding factors, again, positive or negative. The resulting dot product, qi T pu, captures the interaction between user u and item i—the user’s overall interest in the item’s characteristics.



This approximates user u's rating of item i, which is denoted by rui, leading to the estimate

$$\hat{r}_{ui} = q_i^T p_u.$$

The major challenge is computing the mapping of each item and user to factor vectors qi, pu ∈ ・ f. After the recommender system completes this mapping, it can easily estimate the rating a user will give to any item by using Equation above. Such a model is closely related to singular value decomposition (SVD), a well-established technique for identifying latent semantic factors in information retrieval. Applying SVD in the collaborative filtering domain requires factoring the user-item rating matrix. This often raises difficulties due to the high portion of missing values caused by sparseness in the user-item ratings matrix. Conventional SVD is undefined when knowledge about the matrix is incomplete.

Earlier systems relied on imputation to fill in missing ratings and make the rating matrix dense. However, imputation can be very expensive as it significantly increases the amount of data. In addition, inaccurate imputation might distort the data considerably. Hence, more recent works suggested modeling directly the observed ratings only, while avoiding overfitting through a regularized model. To learn the factor vectors (pu and qi), the system minimizes the regularized squared error on the set of known ratings:

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

Here, κ is the set of the (u,i) pairs for which rui is known (the training set).

The system learns the model by fitting the previously observed ratings. However, the goal is to generalize those previous ratings in a way that predicts future, unknown ratings. Thus, the system should avoid overfitting the observed data by regularizing the learned parameters, whose magnitudes are penalized. The constant λ controls the extent of regularization and is usually determined by cross-validation.

3.2.1 Implicit matrix factorization

So far, we have dealt with explicit preferences such as ratings. However, much of the preference data that we might be able to collect is implicit feedback, where the preferences between a user and item are not given to us, but are, instead, implied from the interactions they might have with an item. Examples include binary data (such as whether a user viewed a movie, whether they purchased a product, and so on) as well as count data (such as the number of times a user watched a movie). There are many different approaches to deal with implicit data. MLlib implements a particular approach that treats the input rating matrix as two matrices: a binary preference matrix, P, and a matrix of confidence weights, C.

For example, let's assume that the user-movie ratings were, in fact, the number of times each user had viewed that movie. Here, the matrix P informs us that a movie was viewed by a user, and the matrix C represents the confidence weighting, in the form of the view counts—generally, the more a user has watched a movie, the higher the confidence that they actually like it.

The implicit model still creates a user- and item-factor matrix. In this case, however, the matrix that the model is attempting to approximate is not the overall ratings matrix but the preference matrix P. If we compute a recommendation by calculating the dot product of a user- and item-factor vector, the score will not be an estimate of a rating directly. It will rather be an estimate of the preference of a user for an item (though not strictly between 0 and 1, these scores will generally be fairly close to a scale of 0 to 1).

3.2.2 Alternating least squares

Because both qi and pu are unknowns, Equation above is not convex. However, if we fix one of the unknowns, the optimization problem becomes quadratic and can be solved optimally. Thus, ALS techniques rotate between fixing the qi's and fixing the pu's. When all pu's are fixed, the system recomputes the qi's by solving a least-squares problem, and vice versa. This ensures that each step decreases the equation until convergence.
While in general stochastic gradient descent is easier and faster than ALS, ALS is favorable in at least two cases. The first is when the system can use parallelization. In ALS, the system computes each qi independently of the other item factors and computes each pu independently of the other user factors. This gives rise to potentially massive parallelization of the algorithm. The second case is for systems centered on implicit data. Because the training set cannot be considered sparse, looping over each single training case—as gradient descent does—would not be practical. ALS can efficiently handle such cases.

Collaborative filtering approaches often suffer from three problems: cold start, scalability, and sparsity.

Cold Start: These systems often require a large amount of existing data on a user in order to make accurate recommendations.

Scalability: In many of the environments in which these systems make recommendations, there are millions of users and products. Thus, a large amount of computation power is often necessary to calculate recommendations.

Sparsity: The number of items sold on major e-commerce sites is extremely large. The most active users will only have rated a small subset of the overall database. Thus, even the most popular items have very few ratings.

Collaborative filtering techniques are classified as memory-based and model based collaborative filtering. A well-known example of memory-based approaches is user-based algorithm and that of model-based approaches is Kernel-Mapping Recommender.

3.3 Content-based filtering

Another common approach when designing recommender systems is content-based filtering. Content-based filtering methods are based on a description of the item and a profile of the user’s preference. In a content-based recommender system, keywords are used to describe the items; beside, a user profile is built to indicate the type of item this user likes. In other words, these algorithms try to recommend items that are similar to those that a user liked in the past (or is examining in the present). In particular, various candidate items are compared with items previously rated by the user and the best-matching items are recommended. This approach has its roots in information retrieval and information filtering research.

To create user profile, the system mostly focuses on two types of information:

1. A model of the user's preference.

2. A history of the user's interaction with the recommender system.

Basically, these methods use an item profile (i.e. a set of discrete attributes and features) characterizing the item within the system. The system creates a content-based profile of users based on a weighted vector of item features. The weights denote the importance of each feature to the user and can be computed from individually rated content vectors using a variety of techniques. Simple approaches use the average values of the rated item vector while other sophisticated methods use machine learning techniques such as Bayesian Classifiers, cluster analysis, decision trees, and artificial neural networks in order to estimate the probability that the user is going to like the item.

Direct feedback from a user, usually in the form of a like or dislike button, can be used to assign higher or lower weights on the importance of certain attributes (using Rocchio Classification or other similar techniques).

A key issue with content-based filtering is whether the system is able to learn user preferences from user's actions regarding one content source and use them across other content types. When the system is limited to recommending content of the same type as the user is already using, the value from the recommendation system is significantly less than when other content types from other services can be recommended. For example, recommending news articles based on browsing of news is useful, but it's much more useful when music, videos, products, discussions etc. from different services can be recommended based on news browsing.

3.4 Hybrid Recommender Systems

Recent research has demonstrated that a hybrid approach, combining collaborative filtering and content-based filtering could be more effective in some cases. Hybrid approaches can be implemented in several ways: by making content-based and collaborative-based predictions separately and then combining them; by adding content-based capabilities to a collaborative-based approach (and vice versa); or by unifying the approaches into one model. Several studies empirically compare the performance of the hybrid with the pure collaborative and content-based methods and demonstrate that the hybrid methods can provide more accurate recommendations than pure approaches. These methods can also be used to overcome some of the common problems in recommender systems such as cold start and the sparsity problem.

In hybrid systems recommendations are made 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).

A variety of techniques have been proposed as the basis for recommender systems: collaborative, content-based, knowledge-based, and demographic techniques. Each of these techniques has known shortcomings, such as the well known cold-start problem for collaborative and content-based systems (what to do with new users with few ratings) and the knowledge engineering bottleneck in knowledge-based approaches. A hybrid recommender system is one that combines multiple techniques together to achieve some synergy between them.

Collaborative: The system generates recommendations using only information about rating profiles for different users. Collaborative systems locate peer users with a rating history similar to the current user and generate recommendations using this neighborhood.

Content-based: The system generates recommendations from two sources: the features associated with products and the ratings that a user has given them. Content-based recommenders treat recommendation as a user-specific classification problem and learn a classifier for the user's likes and dislikes based on product features.

Demographic: A demographic recommender provides recommendations based on a demographic profile of the user. Recommended products can be produced for different demographic niches, by combining the ratings of users in those niches.

Knowledge-based: A knowledge-based recommender suggests products based on inferences about a user's needs and preferences. This knowledge will sometimes contain explicit functional knowledge about how certain product features meet user needs.

The term hybrid recommender system is used here to describe any recommender system that combines multiple recommendation techniques together to produce its output. There is no reason why several different techniques of the same type could not be hybridized, for example, two different content-based recommenders could work together. Below are few hybridization techniques:

Weighted: The score of different recommendation components are combined numerically.

Switching: The system chooses among recommendation components and applies the selected one.

Mixed: Recommendations from different recommenders are presented together.

Feature Combination: Features derived from different knowledge sources are combined together and given to a single recommendation algorithm.

Feature Augmentation: One recommendation technique is used to compute a feature or set of features, which is then part of the input to the next technique.

Cascade: Recommenders are given strict priority, with the lower priority ones breaking ties in the scoring of the higher ones.

Meta-level: One recommendation technique is applied and produces some sort of model, which is then the input used by the next technique.

4 Parallel Programming with Apache Spark

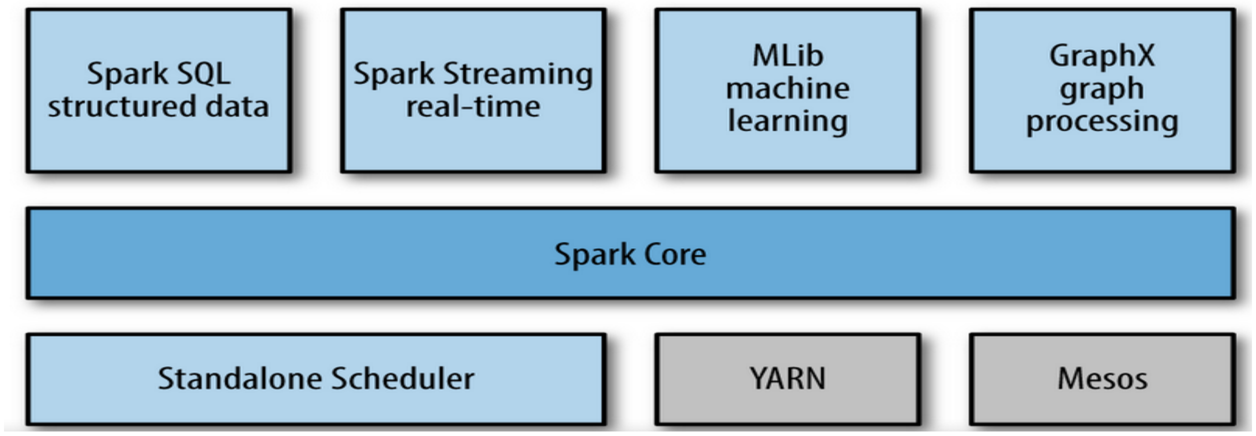
Apache Spark is a cluster computing platform designed to be *fast* and *general-purpose*.

On the speed side, Spark extends the popular MapReduce model to efficiently support more types of computations, including interactive queries and stream processing. Speed is important in processing large datasets, as it means the difference between exploring data interactively and waiting minutes or hours. One of the main features Spark offers for speed is the ability to run computations in memory, but the system is also more efficient than MapReduce for complex applications running on disk.

On the generality side, Spark is designed to cover a wide range of workloads that previously required separate distributed systems, including batch applications, iterative algorithms, interactive queries, and streaming. By supporting these workloads in the same engine, Spark makes it easy and inexpensive to *combine* different processing types, which is often necessary in production data analysis pipelines. In addition, it reduces the management burden of maintaining separate tools.

Spark is designed to be highly accessible, offering simple APIs in Python, Java, Scala, and SQL, and rich built-in libraries. It also integrates closely with other Big Data tools. In particular, Spark can run in Hadoop clusters and access any Hadoop data source, including Cassandra.

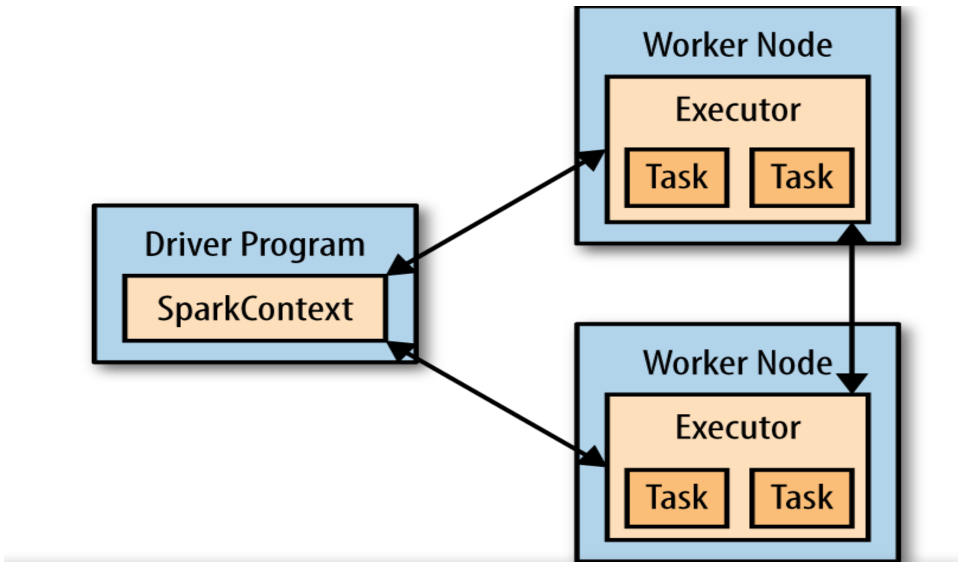
The Spark project contains multiple closely integrated components. At its core, Spark is a "computational engine" that is responsible for scheduling, distributing, and monitoring applications consisting of many computational tasks across many worker machines, or a *computing cluster*. Because the core engine of Spark is both fast and general-purpose, it powers multiple higher-level components specialized for various workloads, such as SQL or machine learning. These components are designed to interoperate closely, letting one combine them like libraries in a software project.



4.1 Introduction to Core Spark Concepts

4.1.1 Spark Clusters

A Spark cluster is made up of two types of processes: a driver program and multiple executors. In the local mode, all these processes are run within the same JVM. In a cluster, these processes are usually run on separate nodes.
For example, a typical cluster that runs in Spark's standalone mode (that is, using Spark's built-in cluster-management modules) will have:
- A master node that runs the Spark standalone master process as well as the driver program
- A number of worker nodes, each running an executor process



4.1.2 Programming model

At a high level, every Spark application consists of a *driver program* that launches various parallel operations on a cluster. The driver program contains application's main function and defines distributed datasets on the cluster, then applies operations to them. By caching the data in memory on each worker node, Spark ultimately reduces amount of data being passed over the network and as result is especially well-suited to iterative algorithms which reuse a working set of data across multiple parallel algorithms. In these circumstances, the reduced amount of data passed over the network enables Spark applications to run significantly faster when compared to disk-based implementations on Hadoop. In certain cases, Spark applications have achieved up to a 10x speed increase over Hadoop, although this result is highly dependent on the nature of the underlying algorithm.
As a bi-product of Spark's data model, iterative algorithms that cannot be expressed well in the two-stage MapReduce programming model are more easily implemented on Spark. Furthermore, the use of in-memory caching of partitioned data on each worker node enables fast, interactive analysis of big datasets. Frameworks such as Pig and Hive, which are built on top of Hadoop and used to perform ad-hoc exploratory data analysis, incur significant latency when compared to Spark from having to read data from disk with each MapReduce job.

Implementing parallel programs in Spark is accomplished through the use of a driver program that runs on the master node of a Spark cluster and is responsible for expressing the high-level control flow of a Spark job. Within a driver program, a Spark user defines a series of operations, such as map, reduce, and filter, which are executed on each of the worker nodes in the cluster. This dataflow programming model is accomplished through the use of several core abstractions provided by the Spark framework,most notably resilient distributed datasets, parallel operations, and shared variables, which are discussed below in detail.

Driver programs access Spark through a SparkContext object, which represents a connection to a computing cluster.

The resilient distributed dataset (RDD) is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. In a Spark program, data is first read in from a distributed file system, such as the Hadoop Distributed File System (HDFS), into an RDD object. The RDD is then parallelized by the driver program, causing it to be partitioned and sent to multiple nodes. RDDs are lazy and ephemeral by default, meaning that the data in each partition only becomes available when used in a parallel operation and is discarded from memory after use. However, for data that is intended to be reused later on in the algorithm, the persistence on an RDD can be altered by the use of a cache action, which advises Spark to keep the RDD in memory of the worker nodes to improve performance.

Once data has been read into an RDD, the initialized RDD can be transformed
by applying parallel operations to it. For example, a parallel operation might be used to pass each element in an RDD object through a user-defined function via the map operation. If the user-defined function transforms values of type X to values of type Y, then each element in the RDD will consist of values of type Y after the operation is applied.

RDDs support two types of operations: transformations and actions.
Transformations: These are operations on RDDs that return a new RDD, such as map() and filter().
Actions: These are operations that return a result to the driver program or write it to storage, and kick off a computation, such as count() and first().

Spark treats transformations and actions very differently, so understanding which type of operation one is performing will be important.

4.1.3 Lazy Evaluation

Transformations are operations on RDDs that return a new RDD. As discussed in "Lazy Evaluation", transformed RDDs are computed lazily, only when we use them in an action. Many transformations are element-wise; that is, they work on one element at a time; but this is not true for all transformations.

Spark offers a number of different parallel operations which can be applied on RDDs; some them that are used in the context of collaborative filtering algorithm are discussed as below.

map: Passes each element in the RDD through a user-defined function. Each element in the resulting RDD is the output of this function applied on the element in the original RDD filter. Passes each element in the RDD through a user-defined predicate function which returns a boolean value. Each element in the resulting RDD is made up of elements in the original RDD for which the predicate function returns True.

groupByKey: Aggregates the elements in an RDD based on the key of each element. When called on an RDD consisting of (K,v) pairs, returns all (K,Seq[V]) pairs where each key in the resulting RDD is unique.

collect: Sends all elements in the RDD to the driver program, to be used when user wants to collect the results in the master node. One of the most powerful features of Spark is the ability to cache data in memory across a cluster. This is achieved through use of the cache method on an RDD:

cache: Calling cache on an RDD tells Spark that the RDD should be kept in memory. The first time an action is called on the RDD that initiates a computation, the data is read from its source and put into memory. Hence, the first time such an operation is called, the time it takes to run the task is partly dependent on the time it takes to read the data from the input source. However, when the data is accessed the next time (for example, in subsequent queries in analytics or iterations in a machine learning model), the data can be read directly from memory, thus avoiding expensive I/O operations and speeding up the computation, in many cases, by a significant factor.

Lastly, Spark also enables the use of shared variables, such as broadcast and ac-culumulator variables for accessing or updating shared data accross worker nodes. Shared variables are copied to each worker node, and no updates to the variables on those nodes are propagated back to the driver program. In specific, if a large read-only piece of data is accessed in multiple parallel operations, using a broadcast variable is more efficient than storing that data in a read-write variable across tasks, since the data associated with that object is ensured to be only shipped to each worker once. Although the data in a parallelized RDD object is cached on each worker node already, the use of a broadcast variable prevents having to package the data stored in it with every task, thus increasing the efficiency of the Spark application.

4.2 Machine learning with Mllib

MLlib is Spark's library of machine learning functions. Designed to run in parallel on clusters, MLlib contains a variety of learning algorithms and is accessible from all of Spark's programming languages. MLlib's design and philosophy are simple: it lets us invoke various algorithms on distributed datasets, representing all data as RDDs. One important thing to note about MLlib is that it contains only parallel algorithms that run well on clusters. Some classic ML algorithms are not included because they were not designed for parallel platforms, but in contrast MLlib contains several recent research algorithms for clusters, such as distributed random forests, K-means||, and alternating least squares. This choice means that MLlib is best suited for running each algorithm on a large dataset. If we instead have many small datasets on which we want to train different learning models, it would be better to use a single-node learning library (e.g., Weka or SciKit-Learn) on each node, perhaps calling it in parallel across nodes using a Spark map(). Likewise, it is common for machine learning pipelines to require training the same algorithm on a small dataset with many configurations of parameters, in order to choose the best one. We can achieve this in Spark by using parallelize() over the list of parameters to train different ones on different nodes, again using a single-node learning library on each node. But MLlib itself shines when we have a large, distributed dataset that we need to train a model on.

MLlib requires some linear algebra libraries to be installed on our machines. First, we will need the gfortran runtime library for our operating system. Second, to use MLlib in Python, we will need NumPy. If Python installation does not have it (i.e., we cannot import numpy), the easiest way to get it is by installing the python-numpy or numpy package through package manager on Linux.

From among the key algorithms availalble in Mllib, below we have outlined a brief description of the API corresponding to the collaborative filtering recommender algorithm and Statistics.

4.2.1 Statistics

Basic statistics are an important part of data analysis, both in ad hoc exploration and understanding data for machine learning. MLlib offers several widely used statistic functions that work directly on RDDs, through methods in the mllib.stat.Statistics class. Some commonly used ones include:

Statistics.colStats(rdd): Computes a statistical summary of an RDD of vectors, which stores the min, max, mean, and variance for each column in the set of vectors. This can be used to obtain a wide variety of statistics in one pass.

Statistics.corr(rdd, method): Computes the correlation matrix between columns in an RDD of vectors, using either the Pearson or Spearman correlation (method must be one of pearson and spearman).

Statistics.corr(rdd1, rdd2, method): Computes the correlation between two RDDs of floating-point values, using either the Pearson or Spearman correlation (method must be one of pearson and spearman).

Statistics.chiSqTest(rdd): Computes Pearson's independence test for every feature with the label on an RDD of LabeledPointobjects. Returns an array of ChiSqTestResult objects that capture the p-value, test statistic, and degrees of freedom for each feature. Label and feature values must be categorical (i.e., discrete values). Apart from these methods, RDDs containing numeric data offer several basic statistics such as mean(), stdev(), and sum(), as described in "Numeric RDD Operations". In addition, RDDs support sample() and sampleByKey() to build simple and stratified samples of data.

4.2.2 Collaborative Filtering and Recommendation

Collaborative filtering is a technique for recommender systems wherein users' ratings and interactions with various products are used to recommend new ones. Collaborative filtering is attractive because it only needs to take in a list of user/product interactions: either "explicit" interactions (i.e., ratings on a shopping site) or "implicit" ones (e.g., a user browsed a product page but did not rate the product). Based solely on these interactions, collaborative filtering algorithms learn which products are similar to each other (because the same users interact with them) and which users are similar to each other, and can make new recommendations.

While the MLlib API talks about "users" and "products," one can also use collaborative filtering for other applications, such as recommending users to follow on a social network, tags to add to an article, or songs to add to a radio station.

4.2.2.1 Alternating Least Squares

MLlib includes an implementation of Alternating Least Squares (ALS), a popular algorithm for collaborative filtering that scales well on clusters.22 It is located in the mllib.recommendation.ALS class. ALS works by determining a feature vector for each user and product, such that the dot product of a user's vector and a product's is close to their score. It takes the following parameters:

Rank: Size of feature vectors to use; larger ranks can lead to better models but are more expensive to compute (default: 10).

Iterations: Number of iterations to run (default: 10).

Lambda: Regularization parameter (default: 0.01).

Alpha: A constant used for computing confidence in implicit ALS (default: 1.0).

numUserBlocks, numProductBlocks: Number of blocks to divide user and product data in, to control parallelism; one can pass –1 to let MLlib automatically determine this (the default behavior).

To use ALS, one need to give it an RDD of mllib.recommendation.Rating objects, each of which contains a user ID, a product ID, and a rating (either an explicit rating or implicit feedback; see upcoming discussion). One challenge with the implementation is that each ID needs to be a 32-bit integer. If IDs are strings or larger numbers, it is recommended to just use the hash code of each ID in ALS; even if two users or products map to the same ID, overall results can still be good. Alternatively, one can broadcast() a table of product-ID-to-integer mappings to give them unique IDs.

ALS returns a MatrixFactorizationModel representing its results, which can be used to predict() ratings for an RDD of (userID, productID) pairs. Alternatively, one can use model.recommendProducts(userId, numProducts) to find the top numProducts() products recommended for a given user. Note that unlike other models in MLlib, the MatrixFactorizationModel is large, holding one vector for each user and product. This means that it cannot be saved to disk and then loaded back in another run of program. Instead, we can save the RDDs of feature vectors produced in it, model.userFeatures and model.productFeatures, to a distributed filesystem.

Finally, there are two variants of ALS: for explicit ratings (the default) and for implicit ratings (which we enable by calling ALS.trainImplicit() instead of ALS.train()). With explicit ratings, each user's rating for a product needs to be a score (e.g., 1 to 5 stars), and the predicted ratings will be scores. With implicit feedback, each rating represents a confidence that users will interact with a given item (e.g., the rating might go up the more times a user visits a web page), and the predicted items will be confidence values.

5 Recommendation system with Apache Spark

Considering the case of requirement to recommend TV programs for users from among several hundreds of available programs everyday so that users are able to easily find the interesting ones and watch such programs at the preferred time of the day, we start with taking data that will required as part of machine learning pipe line.

5.1 Exploring and preparing Data

The system assumes that user registers with it by creating a profile with the system. Uses are invited to specify some profile details, in particular, viewed channels, programme titles, content keywords, and preferred viewing times, employment status, marital status , geographic location etc . This provides system with a starting point for each user, and these preliminary profiles are elaborated upon and refined as system learns more about the likes and dislikes of individual users.

On top of this the user activity is continuously monitored by the system and logs various parameters pertaining to the usage of the system. This includes the channels they watch; programme titles; content keywords; preferred viewing times; a list of positively graded programmes; a list of negatively graded programmes; a genre schema; various administrative details such as login times, duration of watching

Each program is described by a set of fields describing data such as the starting time of the program, the transmission channel and the stream content (video, audio or data). The program categories are organized in taxonomy, the General Ontology, which includes several broad categories, such as Serial, and specializes such categories into more specific ones, such as Soap Opera, Science Fiction Serial, etc. The program database includes programe cases where in each entry describes a particular programme using features such as the programme title, genre information, the creator and director, cast or presenters, the country of origin, and the language.

This systems schedule database also contains TV listings for all supported channels. Each listing entry includes details such as the programme name, the viewing channel, the start and end time, and typically some text describing the programme in question The schedule database is constructed automatically from online schedule resources (e.g., online teletext pages and static entertainment guides) by system's schedule agents. Each agent is designed to mine a particular online resource for relevant schedule information and the results of these many parallel searches is the compilation of a rich schedule database

Also the other forms of data include a prior information about the preferences of stereotypical classes of TV viewers.

With data available in various forms as above during preprocessing phase, it becomes critical to establish the key goal of managing this data in terms of contextual information already available.

"To recommend items to a given user by defining the data model considering his/her interests within a period of a day and viewing preferences based on the rating provided for programs as per this watching habits"

Once we have performed initial preprocessing on our data, we often need to transform the data into a representation that is suitable for machine learning models. For many model types, this representation will take the form of a vector or matrix structure that contains numerical data and identifiers for the users and products. Before the actual transformation and further to preprocessing extracting useful features from among the various relations with the user – item requires to be performed.

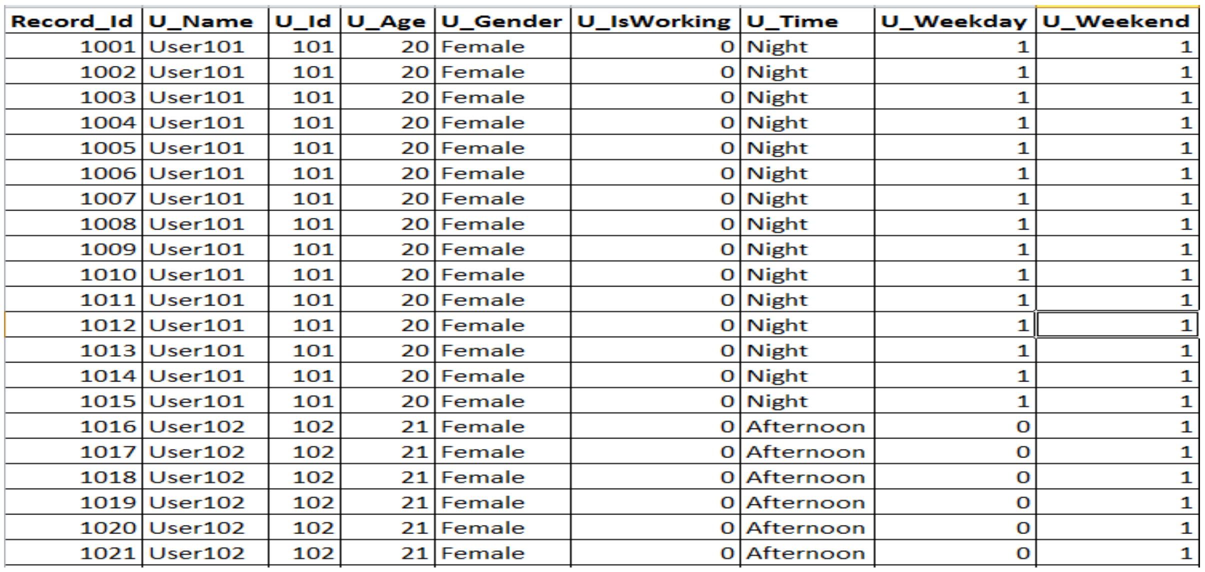
The following list of feature attributes were found to be significant considering a user –item relation and have been initially extracted from among the various sources of data available.

Feature	**Description**
U_Name	Name of User
U_Id	UserId
U_Age	Age of User
U_Gender	User's Gender
U_IsWorking	Is User a Working professional or Household
U_Time	User's preferred time in the day to watch TV
U_Weekday	Does User watch TV during week days
U_Weekend	Does User watch TV during week ends
P_Name	Program Name
P_Id	Program Id
P_Category	Program Category
P_Channel	TV Channel
P_Genre	Programe Genre
P_SubGenre	Programe Sub-Genre
P_IsCastFamous	Is Program's Cast famous
P_Weekend	Is Program scheduled for weekend
P_Weekday	Is Program scheduled for weekend
P_3MonthSeries	Is Program a 3 Month Series
P_3YearSeries	Is Program a 3 Year Series
P_Time	Program view time
P_Rating	Rating for the provided by user to a programme

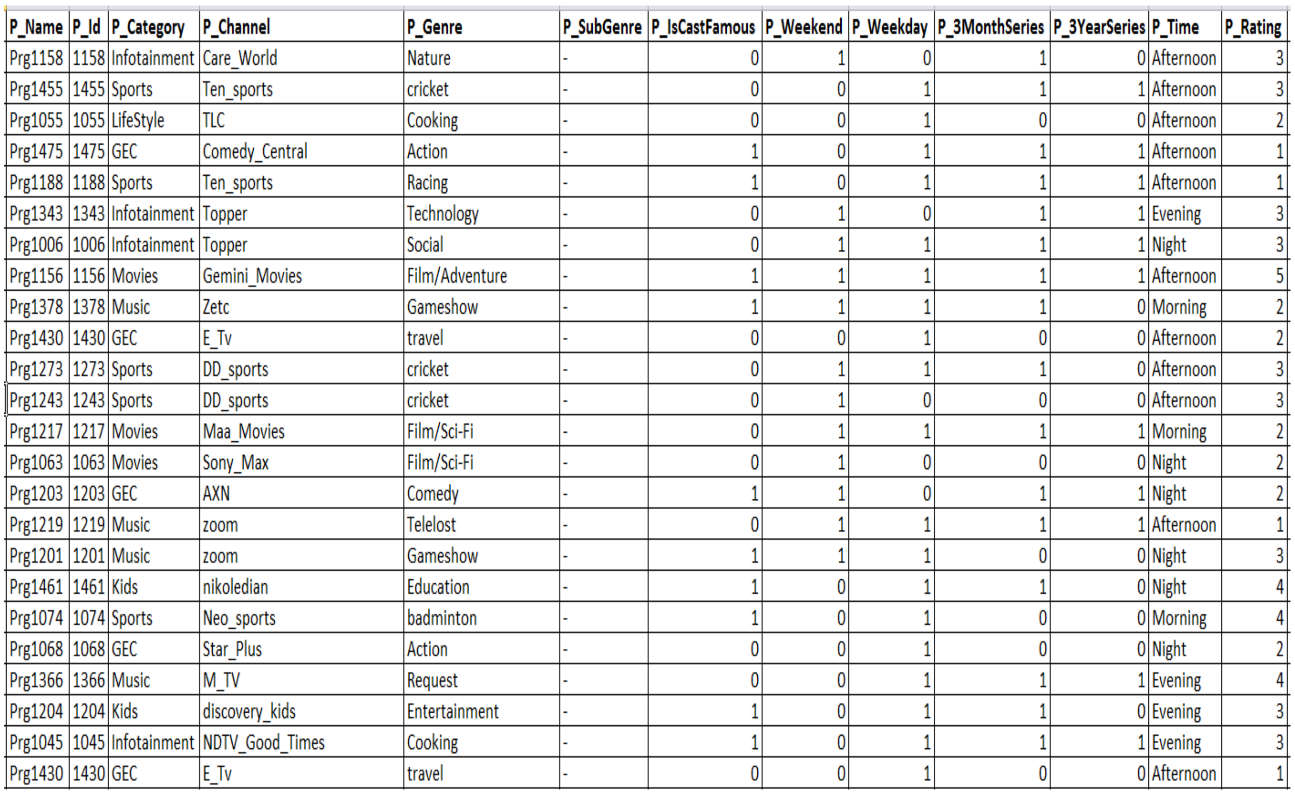
For such a relational set of features, data is generated for about 1000 user profiles and 500 programmes. A java program (**TVListings.java in Appendix D**) is has been created to generate data into a csv with each line representing a single user-item relation.

The generated data file (**up_data.csv** in **Appendix D**) contains watching history for several users relating their profile with programs they watched at different times within a day (weekend/weekday) along with programmes attributes categorized by Genre, repeating series as per the features identified above.

This data is further cleansed, transformed to arrive at corresponding string/numeric representation of each feature. Over a process of feature engineering further dependencies between the various features and their influence on the requirement to recommend tv programmes is evaluated to arrive at data is suitable for the model training.



Record_Id	U_Name	U_Id	U_Age	U_Gender	U_IsWorking	U_Time	U_Weekday	U_Weekend
1001	User101	101	20	Female	0	Night	1	1
1002	User101	101	20	Female	0	Night	1	1
1003	User101	101	20	Female	0	Night	1	1
1004	User101	101	20	Female	0	Night	1	1
1005	User101	101	20	Female	0	Night	1	1
1006	User101	101	20	Female	0	Night	1	1
1007	User101	101	20	Female	0	Night	1	1
1008	User101	101	20	Female	0	Night	1	1
1009	User101	101	20	Female	0	Night	1	1
1010	User101	101	20	Female	0	Night	1	1
1011	User101	101	20	Female	0	Night	1	1
1012	User101	101	20	Female	0	Night	1	1
1013	User101	101	20	Female	0	Night	1	1
1014	User101	101	20	Female	0	Night	1	1
1015	User101	101	20	Female	0	Night	1	1
1016	User102	102	21	Female	0	Afternoon	0	1
1017	User102	102	21	Female	0	Afternoon	0	1
1018	User102	102	21	Female	0	Afternoon	0	1
1019	User102	102	21	Female	0	Afternoon	0	1
1020	User102	102	21	Female	0	Afternoon	0	1
1021	User102	102	21	Female	0	Afternoon	0	1



P_Name	P_Id	P_Category	P_Channel	P_Genre	P_SubGenre	P_IsCastFamous	P_Weekend	P_Weekday	P_3MonthSeries	P_3YearSeries	P_Time	P_Rating
Prg1158	1158	Infotainment	Care_World	Nature	-	0	1	0	1	0	Afternoon	3
Prg1455	1455	Sports	Ten_sports	cricket	-	0	0	1	1	1	Afternoon	3
Prg1055	1055	LifeStyle	TLC	Cooking	-	0	0	1	0	0	Afternoon	2
Prg1475	1475	GEC	Comedy_Central	Action	-	1	0	1	1	1	Afternoon	1
Prg1188	1188	Sports	Ten_sports	Racing	-	1	0	1	1	1	Afternoon	1
Prg1343	1343	Infotainment	Topper	Technology	-	0	1	0	1	1	Evening	3
Prg1006	1006	Infotainment	Topper	Social	-	0	1	1	1	1	Night	3
Prg1156	1156	Movies	Gemini_Movies	Film/Adventure	-	1	1	1	1	1	Afternoon	5
Prg1378	1378	Music	Zetc	Gameshow	-	1	1	1	1	0	Morning	2
Prg1430	1430	GEC	E_Tv	travel	-	0	0	1	0	0	Afternoon	2
Prg1273	1273	Sports	DD_sports	cricket	-	0	1	1	1	0	Afternoon	3
Prg1243	1243	Sports	DD_sports	cricket	-	0	1	0	0	0	Afternoon	3
Prg1217	1217	Movies	Maa_Movies	Film/Sci-Fi	-	0	1	1	1	1	Morning	2
Prg1063	1063	Movies	Sony_Max	Film/Sci-Fi	-	0	1	0	0	0	Night	2
Prg1203	1203	GEC	AXN	Comedy	-	1	1	0	1	1	Night	2
Prg1219	1219	Music	zoom	Telelost	-	0	1	1	1	1	Afternoon	1
Prg1201	1201	Music	zoom	Gameshow	-	1	1	1	0	0	Night	3
Prg1461	1461	Kids	nikoledian	Education	-	1	0	1	1	0	Night	4
Prg1074	1074	Sports	Neo_sports	badminton	-	1	0	1	0	0	Morning	4
Prg1068	1068	GEC	Star_Plus	Action	-	0	0	1	0	0	Night	2
Prg1366	1366	Music	M_TV	Request	-	0	0	1	1	1	Evening	4
Prg1204	1204	Kids	discovery_kids	Entertainment	-	1	0	1	1	0	Evening	3
Prg1045	1045	Infotainment	NDTV_Good_Times	Cooking	-	1	0	1	1	1	Evening	3
Prg1430	1430	GEC	E_Tv	travel	-	0	0	1	0	0	Afternoon	1

The feature attributes as in the below table are found to be suitable for processing the data required for training the model.

Feature	**Description**	**Type**
U_Name	Name of User	String
U_Id	User Id	Integer
P_Name	Program Name	String
P_Id	Program Id	Integer
P_Category	Program Category	String
P_Channel	TV Channel	String
P_Genre	Programe Genre	String
P_IsCastFamous	Is Program's Cast famous	Integer {0,1}
P_Weekend	Is Program scheduled for weekend	Integer {0,1}
P_Weekday	Is Program scheduled for weekend	Integer {0,1}
P_3MonthSeries	Is Program a 3 Month Series	Integer {0,1}
P_3YearSeries	Is Program a 3 Year Series	Integer {0,1}
P_Time	Program view time	String {'Morning','Afternoon','Evening','Night' }
P_Rating	Rating provided by user for a given programme	Integer {1,2,3,4,5}

Considering various forms of implicit feedback possible a ALS based collaborative filtering model is being considered for training the data. As discussed in the previous section content-based capabilities are added to collaborative approach to arrive at a hybrid recommendations model; this seems most desirable model for the current requirement.
Now, considering the requirement to recommend programms, data in terms of feature vectors a holistic solution is considered. The key points of which are as below …

For anonymous user or new users, its the 'most watched' program by existing users of the system. The 'most watched' item is to be given priority, from among the live programs the most watched Genre from among the most watched channels/categories corresponding to that time slot could be suggested.

For existing users of the system, from previous wathcing habits of the users within that time slot programs are sorted in terms of most watched Genre and most watched Category/Channel by that user during that time slot and programs (most watched) or those that are being watched or subscribed by other users who have similar preference of Genre and Channel category during that time slot could be recommended. In case their exists a tv show which is a regular series within the neighbour hood of that time slot and if its part of user's watching history it could be recommended, it could be given a high priority recommendation wise.

The time slot during which the program is broadcasted also has priority. A program can have additional feature (ranking) if its relevant to watch a program live or if it can be watched later at user's convenience later.

'Most Watched' could be a rating factor value and which could be a function of parameters as below:

- Frequency of appearence of a Genre/Category/Program within the list of Programs viewed by a user or a group of users within a defined period of time. In case of ties arbitrary value from most common could be choosed.
- Popularity of the Cast or Event.
- Rating provided by users

From above we noticed that user watching habits of programs given a view time/day of the user and user's feedback an ALS based recommendation system can be modelled using the machine learning capabilities of Apache Spark. On top of this the context and content based capabilities can grouped to provide an overall recommendation. Also from the nature of solution it suggests that an item-item similarity has a significant role to play and can be used for predictions as well based on the ALS modelled user – item factor vectors.

5.2 ALS based recommendation system

The next step is to extract right features from data required for training the model. A python script is created to handle the data operations of reading data from data file, parsing it, extracting the fields of interest into locally cached RDDs, model the data using MLlib API from spark and later provide predictions by user and item-item similarities. This process is briefly described as below.

5.2.1 Setup and execution

Inspect the readiness of the system for running spark jobs by few checks such as verifying the amount of memory available, the setup status of Spark, numpy etc. Move the data files and script file to a path that could be resovled by Spark API. The path to data file is currently passed as a command line parameter

Running the program
The python script (**tvProgReco.py in Appendix D**) could be run by submitting a python job over as below

```
>./bin/spark-submit <path_to_python_script> <path_to_data_folder> <UserId>
```

The userId in the above command line parameter corresponds to that of user who needs to be recommended with items.

Once executed the script loads the "program.csv" file from the data folder path in command line argument list and parses the same to load the key features.

The 'ParseData' method parses each line of the file by "," character and return a collection of fields for the data and stores the data in a data table format in memory into a local RDD variable. Later other variables apply transformations such as filter and map to extract the features required out of this collection of data.

Every time the script calculates the time context and broadly classifies it as one among the items in the list below:

['Morning','Afternoon','Evening','Night']

Similarly another variable calculates the weekday when the recommendation is generated and transforms this information into a binary flag to filter data accordingly for weekdays and weekends.

The output from program is redirected to a log file (**output_log.txt in Appendix D**) for further analysis.

5.2.2 Training using ALS

We will use MLlib's ALS to train a MatrixFactorizationModel, which takes a RDD[Rating] object as input in Scala and RDD[(user, product, rating)] in Python. ALS has training parameters such as rank for matrix factors and regularization constants. To determine a good combination of the training parameters, we split the data into three non-overlapping subsets, named training, test, and validation, based on the last digit of the timestamp, and cache them.

We will train multiple models based on the training set, select the best model on the validation set based on RMSE (Root Mean Squared Error), and finally evaluate the best model on the test set. We also add personal ratings to the training set to make personalized recommendations. We hold the training, validation, and test sets in memory by calling cache because we need to visit them multiple times.
we will use ALS.train to train a bunch of models, and select and evaluate the best. Among the training paramters of ALS, the most important ones are rank, lambda (regularization constant), and number of iterations.

• **Rank**: This refers to the number of factors in our ALS model, that is, the number of hidden features in our low-rank approximation matrices. Generally, the greater the number of factors, the better, but this has a direct impact on memory usage, both for computation and to store models for serving, particularly for large number of users or items. Hence, this is often a trade-off in real-world use cases. A rank in the range of 10 to 200 is usually reasonable.

• **Iterations**: This refers to the number of iterations to run. While each iteration in ALS is guaranteed to decrease the reconstruction error of the ratings matrix, ALS models will converge to a reasonably good solution after relatively few iterations. So, we don't need to run for too many iterations in most cases (around 10 is often a good default).

• **Lambda**: This parameter controls the regularization of our model. Thus, lambda controls over fitting. The higher the value of lambda, the more is the regularization applied. What constitutes a sensible value is very dependent on the size, nature, and sparsity of the underlying data, and as with almost all machine learning models, the regularization parameter is something that should be tuned using out-of-sample test data and cross-validation approaches.

5.2.3 Training a model using implicit feedback data

The standard matrix factorization approach in MLlib deals with explicit ratings. To work with implicit data, we can use the trainImplicit method. It is called in a manner similar to the standard train method. There is an additional parameter, alpha, that can be set (and in the same way, the regularization parameter, lambda, should be selected via testing and cross-validation methods). The alpha parameter controls the baseline level of confidence weighting applied. A higher level of alpha tends to make the model more confident about the fact that missing data equates to no preference for the relevant user-item pair.

We start the training with below input parameters for the ALS

ranks = [4, 8, 12, 16, 20, 24]
lambdas = [0.1, 10.0, 0.01, 0.5]
numIters = [10, 20]

The ALS.train() method returns a MatrixFactorizationModel object, which contains the user and item factors in the form of an RDD of (id, factor) pairs. These are called userFeatures and productFeatures, respectively

We use the provided method 'computeRmse' to compute the RMSE on the validation set for each model. The model with the smallest RMSE on the validation set becomes the one selected and its RMSE on the test set is used as the final metric.

The best model can be compared with a navie baseline that always returns mean rating. The test set is used to perform comparision in this case.
User based prediction

5.2.4 User based prediction

The next step is to make personalized recommendations using 'predict' method of the best model object for the user who requires the recommendation. This is done by generating (userid, itemId) pairs for all programs user hasn't rated.

The predict method can also take an RDD of (user, item) IDs as the input
and will generate predictions for each of these. We can use this method to make predictions for many users and items at the same time.

To generate the top-K recommended items for a user, MatrixFactorizationModel provides a convenience method called recommendProducts. This takes two arguments: user and num, where user is the user ID, and num is the number of items to recommend. It returns the top num items ranked in the order of the predicted score. Here, the scores are computed as the dot product between the user-factor vector and each item-factor vector.

For the relevant itemId, the TV program title is extracted and displayed as recommended list of programs. The list of items being recommended can be further grouped based on their content. If by Genre they are part of the list of Genre user watches during that timeslot in a week, these items are given higher priority to the rest.

5.2.5 Item based prediction

Item recommendations are about answering the following question: for a certain item, what are the items most similar to it? Here, the precise definition of similarity is dependent on the model involved. In most cases, similarity is computed by comparing the vector representation of two items using some similarity measure. Common similarity measures include Pearson correlation and cosine similarity for real-valued vectors and Jaccard similarity for binary vectors

The current MatrixFactorizationModel API does not directly support item-to-item similarity computations. For python scripts we need to import numpy module for mathematical calculations on arrays such as dot product and calculating (norm,2) for a given vector.
This is similar to how the existing predict and recommendProducts methods work, except that we will use cosine similarity as opposed to just the dot product.

Since scores are computed as the dot product between the user-factor vector and each item-factor vector, once we have a list of similar items they can further be predicted relative to the product as per the score obtained for dot product with corresponding product vector.

5.3 Model Evaluation and Results

Evaluation metrics are measures of a model's predictive capability or accuracy. Some are direct measures of how well a model predicts the model's target variable (such as Root Mean Squared Error), while others are concerned with how well the model performs at predicting things that might not be directly optimized in the model but are often closer to what we care about in the real world (such as Mean average precision).

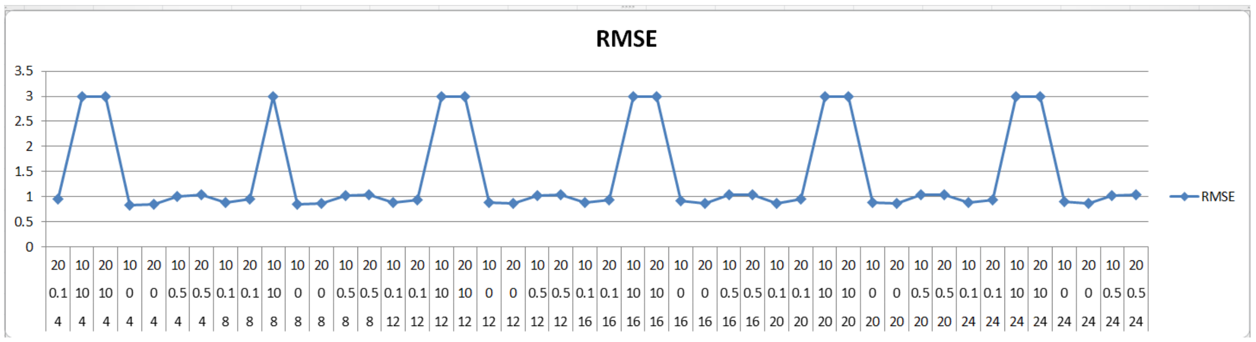
Evaluation metrics provide a standardized way of comparing the performance of the same model with different parameter settings and of comparing performance across different models. Using these metrics, we can perform model selection to choose the best-performing model from the set of models we wish to evaluate.

5.3.1 Mean Squared Error

The Mean Squared Error (MSE) is a direct measure of the reconstruction error of the user-item rating matrix. It is also the objective function being minimized in certain models, specifically many matrix-factorization techniques, including ALS. As such, it is commonly used in explicit ratings settings.
It is defined as the sum of the squared errors divided by the number of observations. The squared error, in turn, is the square of the difference between the predicted rating for a given user-item pair and the actual rating

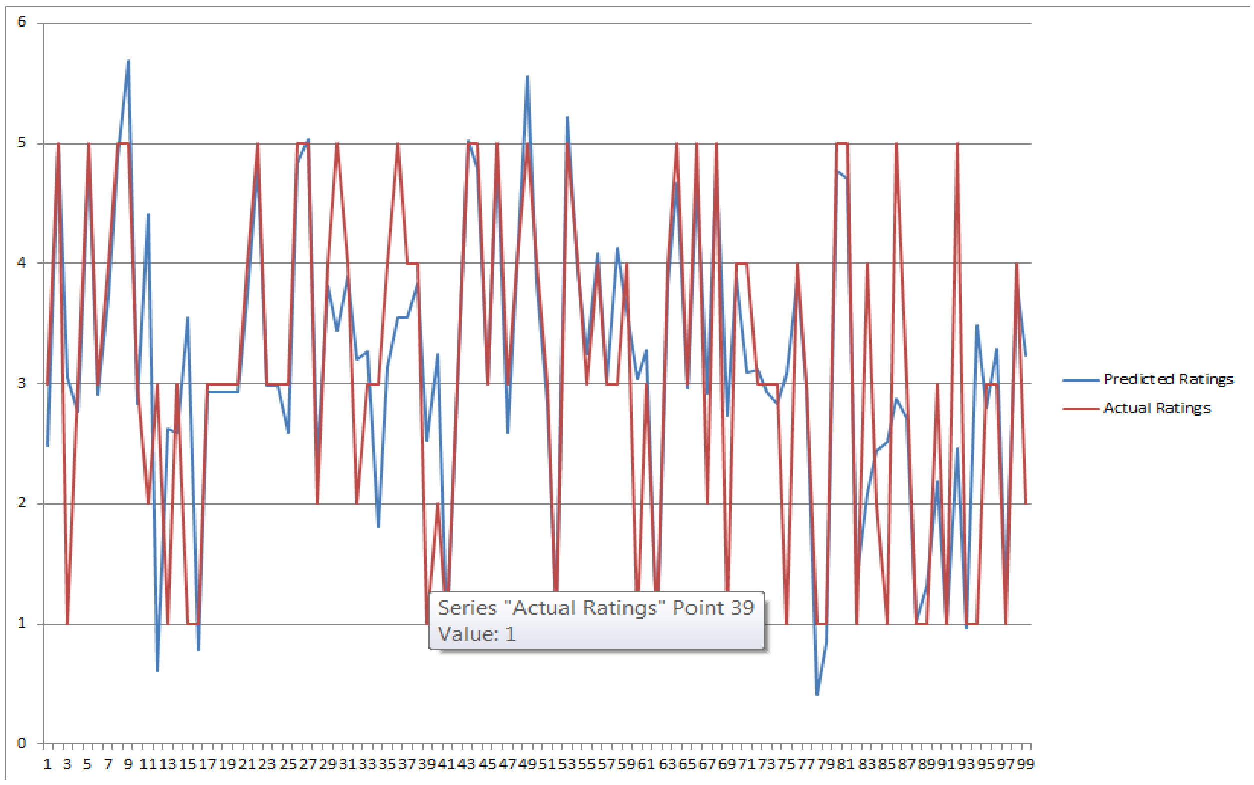
As part of the model evaluation square root of MSE, referred to as RMSE is also used for evaluation of model's accuracy.

Appendix A, lists the RMSE values for varying data for a sample run during weekend and view time as morning for a given userid. This below plot for this data suggests a lower value of lambda leads to more accurate results at the same time a controlled variation of other parameters could result in better results. While an increasing number of iterations does not seem to have major impact on the accuracy of the results.



Plot of Root Mean Square Error for varying values of Rank, Lambda and Iteration Count

Appendix B, lists data corresponding the calculated ratings as per best model and actual ratings for user-item pairs which are part of validation set. The plot of this data, below, provides an overview of variation in terms of accuracy of predicted ratings.



Plot of predicted ratings and actual ratings for set of 100 items within Validation set

Appendix C, lists a set of items recommended by user based prediction and item similarity technique.

Over observation of both the lists with recommended items we find that a calculated union of both the recommended lists as per their priority could lead to a comprehensive list of overall recommended programs. Further study could include analysis on frequency of appearances of an item and its contextual relevance with respect to the recommended items and thus further refining the recommended list.

6 Conclusion

This report has presented a detailed overview of machine learning concepts, recommendation system concepts along with a brief description of distributed computation framework such as Spark. Further to understanding of these building blocks in terms of concept, methodology and technical framework the report considered a case of TV program recommendation for building a recommendation system based on Spark Mllib API. This process comprised of understanding data, cleansing and transforming data so that it is ready for the generating a predictive model. Further a collaborative filtering recommendation model is trained and predictions are made. The predictions made via user based and item based approaches have been considered. The predictive capability of the model is explored via common metrics such as root mean squared error. A hybrid recommendation model comprising a items from both user and item based techniques has been suggested.

Appendix A

Table with RMSE values for corresponding ALS parameters – Rank, Lambda and number of Iterations (Sample run of the recommender system with userId – 104, view time is morning and over weekend.)

RMSE (validation)	**Rank**	**Lambda**	**Number of Iterations**
0.946578	4	0.1	20
2.985791	4	10	10
2.985791	4	10	20
0.827755	4	0.01	10
0.852953	4	0.01	20
1.007933	4	0.5	10
1.033046	4	0.5	20
0.881139	8	0.1	10
0.952596	8	0.1	20
2.985791	8	10	10
0.850214	8	0.01	10
0.871416	8	0.01	20
1.027887	8	0.5	10
1.03739	8	0.5	20
0.891167	12	0.1	10
0.929663	12	0.1	20
2.985791	12	10	10
2.985791	12	10	20
0.878642	12	0.01	10
0.865797	12	0.01	20
1.026322	12	0.5	10
1.042024	12	0.5	20
0.878893	16	0.1	10
0.933993	16	0.1	20
2.985791	16	10	10
2.985791	16	10	20
0.911352	16	0.01	10
0.870049	16	0.01	20
1.031151	16	0.5	10
1.045344	16	0.5	20
0.869378	20	0.1	10
0.945837	20	0.1	20
2.985791	20	10	10
2.985791	20	10	20
0.885265	20	0.01	10

0.869032	20	0.01	20
1.03181	20	0.5	10
1.042483	20	0.5	20
0.884989	24	0.1	10
0.94297	24	0.1	20
2.985791	24	10	10
2.985791	24	10	20
0.904175	24	0.01	10
0.872635	24	0.01	20
1.030498	24	0.5	10
1.04563	24	0.5	20

Appendix B

Table with predicted ratings from best model and actual ratings for corresponding (user-item) pairs (Sample run of the recommender system with UserId – 104, view time is morning and over weekend.)

Predicted Rating	**Actual Rating**
2.479420276	3
4.978918151	5
3.055359181	1
2.771641278	3
4.889799651	5
2.90751997	3
3.725035581	4
4.879192652	5
5.682937657	5
2.833264518	3
4.414837781	2
0.607210567	3
2.62918406	1
2.594698722	3
3.554419731	1
0.778586675	1
2.934955195	3
2.934955195	3
2.934955195	3
2.934955195	3
3.731801538	4
4.814763948	5
2.981917136	3
2.981142606	3
2.587531598	3
4.841594518	5
5.035016405	5
2.353059623	2
3.814449617	4
3.443050187	5
3.887839334	4
3.20096168	2
3.270917323	3
1.807385726	3
3.133490272	4

3.548364931	5
3.549509024	4
3.835466943	4
2.531057743	1
3.250257472	2
0.95642479	1
2.966759002	3
5.022195342	5
4.797808667	5
3.019687223	3
4.869486135	5
2.59062677	3
3.961934068	4
5.552881974	5
3.836668869	4
2.834020413	3
1.096843687	1
5.222420683	5
3.933032221	4
3.245674517	3
4.087374965	4
3.029501105	3
4.128942851	3
3.603166238	4
3.034630503	1
3.27605422	3
0.938106881	1
3.822722925	4
4.678517369	5
2.965995107	3
4.654036944	5
2.924689964	2
4.783917771	5
2.738066626	1
3.881014881	4
3.090595611	4
3.115731613	3
2.927304506	3
2.834388495	3
3.086724407	1
3.871338205	4
2.969926973	3
0.412236212	1

0.847938501	1
4.772825275	5
4.704365327	5
1.386855781	1
2.087232739	4
2.436461964	2
2.514311585	1
2.878293109	5
2.714869037	3
1.02354346	1
1.315871748	1
2.190764299	3
1.038273138	1
2.462561011	5
0.965765025	1
3.481346113	1
2.801390526	3
3.294852059	3
1.23405823	1
3.819726994	4
3.239890112	2
1.912386286	4
3.034630503	3
3.995171874	2
3.905256455	4
2.878293109	3
2.924955601	3
3.447468403	3
2.933622461	3
4.015452216	4
4.018692963	4
1.495311035	3
2.995652487	3
4.979731074	5
1.163159954	1
3.942341374	5
3.150004348	3
2.639467322	3
2.984496814	3
4.421973686	5
2.301899147	5
3.63711212	4
1.022208357	1

3.839545628	4
4.849485451	5
5.361238217	5
2.84629806	4
3.869471705	3
1.761668729	4
4.029138317	4
4.95567917	5
3.023031966	3
5.009526537	5
3.922830973	4
4.965127366	5
4.861585835	5
3.746090317	4
3.130526303	1
3.601051948	3
2.964783883	3
2.990758207	1
3.060343226	3
3.295117045	3
3.820849818	4
3.089040697	3
3.348350113	5
2.717846517	3
0.998010629	3
3.011630501	3
5.214327119	4
3.644715358	4
2.456926712	3
3.478702489	5
2.399013654	1
3.959450336	4
2.933180845	3
4.597221956	5
4.739315552	5
2.907889229	5
4.21720805	5
3.98916729	4
3.008000076	3
2.666159144	3
1.335249016	5
4.083310003	4
1.500465298	2

2.339636588	4
4.579321939	5
2.269228702	4
3.319528201	5
2.456853544	1
4.608942076	5
2.740598711	4
2.721980382	3
3.294739956	3
3.062952571	3
3.744628954	4
4.760254991	5
3.432410563	3
4.019920252	4
2.520825337	4
1.414393572	1
2.038322013	4
1.050110645	1
3.938288788	4
2.922995819	3
2.599504174	4
2.97908126	3
2.001238473	4
5.032986657	5
2.706787839	3
5.036690911	5
2.973954266	3
0.436146364	1
2.903599655	3
4.940454834	5
3.012370676	3
2.951178978	3
2.536648846	3
3.951854766	4
3.400889998	5
4.699365304	5
0.453637172	5
0.869786324	3
3.115170272	3
1.736733424	3
3.731833253	4
3.133720443	4
4.453090052	5

3.011372477	3
2.506011258	3
3.906835935	4
2.99744446	3
4.268567823	5
0.936916171	1
4.514575083	4
3.052103756	4
2.533823446	5
2.848133992	3
3.810302483	4
0.92492098	1
3.953989558	4
2.817626068	3
0.829077479	5
0.917794815	5
0.94988645	1
1.059456881	1
4.817293099	5
3.251116382	4
6.140593435	5
3.922649654	4
2.433002105	2
3.100368491	4
2.789143976	3
2.91303986	3
3.897034374	4
2.152101389	3
4.802852549	5
5.808108301	5
2.424579584	5
2.743023206	3
3.712788023	4
2.970320833	3
3.984386882	4
4.008303747	4
2.446366932	2
0.171467048	2
4.017230851	4
2.446224267	3
2.607163724	3
2.616840354	3
2.91345199	3

2.735631234	3
0.794049542	1
3.982768549	4
3.7706729	4
2.932206236	4
2.908441323	3
1.049591083	4
4.630127199	5
4.736418849	5
0.710790985	3
0.251641374	1
2.905839636	3
3.211665258	4
1.838892507	3
3.870687462	3

Appendix C

Table with list of recommended items for based on user prediction and item similarity methods. (Sample run of the recommender system with userId – 104, view time is morning and over weekend)

Note: The item similarity has been arrived considering a neighborhood of items that are highly rated. The highly rated item by user has been chosen to arrive at the list of similar items.

Item -Item Similarity	User based Prediction
Prg1437	Prg1231
Prg1231	Prg1253
Prg1253	Prg1433
Prg1308	Prg1451
Prg1342	Prg1441
Prg1275	Prg1428
Prg1451	Prg1444
Prg1245	Prg1453
Prg1217	Prg1460
Prg1206	Prg1458
Prg1332	Prg1446
Prg1433	Prg1413
Prg1413	Prg1395
Prg1338	Prg1195
Prg1441	Prg1206
Prg1256	Prg1216
Prg1280	Prg1407
Prg1395	Prg1420
Prg1195	Prg1212
Prg1428	Prg1378
Prg1420	Prg1217
Prg1212	Prg1402
Prg1239	Prg1479
Prg1263	Prg1194
Prg1223	Prg1396
Prg1496	Prg1132
Prg1407	Prg1051
Prg1453	Prg1050
Prg1402	Prg1141
Prg1274	Prg1275
Prg1288	Prg1245

Prg1157	Prg1263
Prg1378	Prg1256
Prg1444	Prg1280
Prg1216	Prg1274
Prg1458	Prg1096
Prg1479	Prg1239
Prg1396	Prg1148
Prg1359	Prg1309
Prg1460	Prg1223
Prg1298	Prg1110
Prg1238	Prg1288
Prg1194	Prg1157
Prg1161	Prg1496
Prg1446	Prg1108
Prg1493	Prg1161
Prg1309	Prg1114
Prg1148	Prg1101
Prg1146	Prg1129
Prg1470	Prg1298

Appendix D

List of project files

1. TVListings.java – Java based program used to generate data
2. data/up_data.csv – Data file in .csv format
3. tvProjReco.py – Python script with implementation of Apache Spark based ALS model
4. logs/output_log.txt – Log file containing the redirected verbose output upon execution of tvp.py. The parameters being {UserId : 104, P_Time : Morning, P_Weekend : 1}

References:

Links to content sources that have been referred as part of current study

1. http://en.wikipedia.org/wiki/Recommender_system
2. http://files.grouplens.org/papers/FnT%20CF%20Recsys%20Survey.pdf
3. http://www2.research.att.com/~volinsky/papers/ieeecomputer.pdf
4. https://www.packtpub.com/big-data-and-business-intelligence/machine-learning-spark
5. http://www.sciencedirect.com/science/article/pii/S1877050913009290
6. http://wanlab.poly.edu/recsys12/recsys/p83.pdf
7. http://research.microsoft.com/pubs/115396/EvaluationMetrics.TR.pdf
8. http://www.ics.uci.edu/~kobsa/courses/ICS206/notes/ardissono.pdf
9. http://aaaipress.org/Papers/IAAI/2000/IAAI00-004.pdf
10. http://shop.oreilly.com/product/0636920028512.do
11. http://www.cs.bme.hu/nagyadat/Recommender_systems_handbook.pdf
12. http://scholarship.claremont.edu/cgi/viewcontent.cgi?article=1914&context=cmc_theses
13. http://www.ccm.media.kyoto-u.ac.jp/~yu/UMUAI_Zhiwen%20Yu.pdf
14. http://ijcai13.org/files/tutorial_slides/td3.pdf
15. https://databricks-training.s3.amazonaws.com/movie-recommendation-with-mllib.html
16. http://www.cs.berkeley.edu/~matei/papers/2010/hotcloud_spark.pdf