Fall 2015

A Recommendation Engine Using Apache Spark

Swapna Kulkarni
San Jose State University

A Recommendation Engine Using Apache Spark

A Project Report

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

By

Swapna Kulkarni

December 2015

The Designated Project Committee Approves the Project Titled

A Recommendation Engine Using Apache Spark

By
Swapna Kulkarni

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY
May 2015

_____
Prof. Duc Thanh Tran Department of Computer Science

_____
Prof. Tsau-Young Lin Department of Computer Science

_____
Prof. James Casaletto Department of Computer Science

ABSTRACT

The volume of structured and unstructured data has grown at exponential scale in recent days. As a result of this rapid data growth, we are always inundated with plethora of choices in any product or service. It is very natural to get lost in the amazon of such choices and finding hard to make decisions. The project aims at addressing this problem by using entity recommendation. The two main aspects that the project concentrates on are implementing and presenting more accurate entity recommendations to the user and another is dealing with vast amount of data. The project aims at presenting recommendation results according to user's query with efficiency and accuracy. Project makes use of ListNet ranking algorithm to rank the recommendation results. Query independent features and query dependent features are used to come up with ranking scores. Ranking scores decide the order in which the recommendation results are presented to the user.

Project makes use of Apache Spark, a distributed big-data processing framework. Spark gives the advantage of handling iterative and interactive algorithms with efficiency and minimal processing time as compared to traditional map-reduce paradigm.

We performed the experiments for recommendation engine using DBPedia as the dataset and tested the results for movie domain. We used both query-independent (pagerank) and query-dependent (click-logs) features for ranking purposes. We observed that ListNet algorithm performs really well by making use of Apache Spark as

the RDDs provide faster way for iterative algorithms to execute. We also observed that the results of recommendation engine are accurate and the entities are well ranked.

ACKNOWLEDGEMENTS

I would like to express my gratitude to Dr. Thanh Tran, my thesis advisor, for his enormous support and motivation in the completion of this project.

I would like to thank my project committee members, Dr. Tsau-Young Lin and Prof. James Casaletto, for their significant contribution towards the completion of this project.

I would like to thank my husband, Hrishikesh Gadre, for his endless encouragement and support throughout my master's studies at San Jose State University.

Table of Contents

List of Figures

List of Tables

1. INTRODUCTION

The outbreak of information in 21st century has lead to overgrowth of data and possible choices that one can have. Which movie should I watch next? Which book should I read? Which link should I click next? We all are inundated with such questions all the times. This decision making process has especially become serious in today's world as people can find everything on Internet.

Entity recommendation systems offer a way of dealing with this vast amount of information and they help users to make the decisions. Entity recommendation can be defined as given an entity under consideration, present/recommend similar entities. Given the large number of related entities in the knowledge base, we need to select the most relevant ones to show based on the current query of the user. Entity ranking helps to retrieve the entities, which are most relevant, based on popularity, authority, relevance etc.

As the volume of data that these entity recommendation systems process is very large, The goal of an entity recommendation system over big data is to design a system that is scalable, efficient, and provides the best possible results for a large variety of queries.

Existing entity recommendation systems use content based filtering, collaborative filtering or knowledge based filtering as recommendation techniques. In content based recommender systems, the recommendation results depend upon the content in the query. These recommender systems create a profile for each product to define its

nature. In case of collaborative filtering, ratings from other users are used for recommendation. Users having similar taste as you are considered for recommendation. This technique has cold start problem, as to begin the algorithm to work, we need the ratings from other users.

This project offers a solution for entity recommendation over Wikipedia data. In the scope of this project, the DBPedia dataset is used. The data is unstructured and each object represents the wikipedia page. The DBpedia links dataset represents links between two wikipedia pages. Relationships between DBpedia resources are constructed using the associated page links between Wikipedia articles.

The DBpedia datasets are used to build an entity graph and get the entities related to entity under consideration. Next problem is to rank the entities. This project uses ListNet algorithm for ranking the entities. Two types of features are taken into consideration. Each entity has a feature vector associated with it. The feature vector contains the values for all the features under consideration. The feature vector is used by the ranking algorithm as an input. This project makes use of Pagerank values for the entity as graph-theoretic feature and and click-log analysis to provide popularity value for the entity.

This project uses ListNet algorithm for training and ranking purposes. The training part results in preparing the training-model which can be used to predict the relevance score for a given DBpedia entity link. As a part of training phase a set of queries and relevant labeled results are prepared and used as an input for the phase. The ranking algorithm uses the training model to output the scores for each of the

results. The results are then sorted in the descending order and the final out consists of top k results.

The system is evaluated with practical datasets, large enough to simulate how professional recommendation engine would work in a minimized scale. Several metrics are tested to compare performances of the chosen strategies and scoring schemes.

The project aims to use the inherent capability of Apache Spark to process the iterative and interactive machine learning algorithms faster than traditional mapreduce model.

2. RELATED WORKS

2.1. Basic Approaches for Recommender Systems

Recommender Systems generally take one of the two approaches: Collaborative Filtering or Content Based Filtering. Some recommender systems also take the hybrid approach of combination of these two approaches.

2.1.1. Collaborative Filtering

In case of this approach, the recommendation is based on model of previous user behaviour. Recommendations given by collaborative filtering are based on automatic collaboration of multiple users and filtered by users with similar tastes. The model can be built based on behaviour of single user or it can be based on behaviour of group of users who have similar taste. When the model is based on group knowledge, it takes into consideration the preferences put out by a group of users who have similar taste as you and based on these preferences, makes a new recommendation.

For example, suppose a recommendation engine for videos on video providing service like YouTube or Netflix is being built. To do so, information from all users who subscribe and use these services can be used. Users with similar preferences can be grouped together. Using this information, most popular video for the group can be decided and can be recommended to other members in that group who have not watched the video.

Following table explains how collaborative filtering is used for video recommendation. The entry in each cell represents how many videos of a particular genre have been watched by particular user.

Video Genre/ Users	Cristina	Preston	Ellen
Comedy	--	9	12
Drama	10	--	15
Sci-Fi	8	11	--

Table 1: Simple example of Collaborative Filtering

Here, this group of users can be clustered together as they have similar interest in many of the video gener. Based on this information, we can recommend Cristina to watch videos under comedy gener, Preston to watch videos under drama genre and Ellen, the Sci-Fi genre. Collaborative filtering can also be defined using similarity-difference approach. Users with similar taste are grouped together and differences in their tastes are potential areas for recommendation. Following Venn Diagram explains the same.

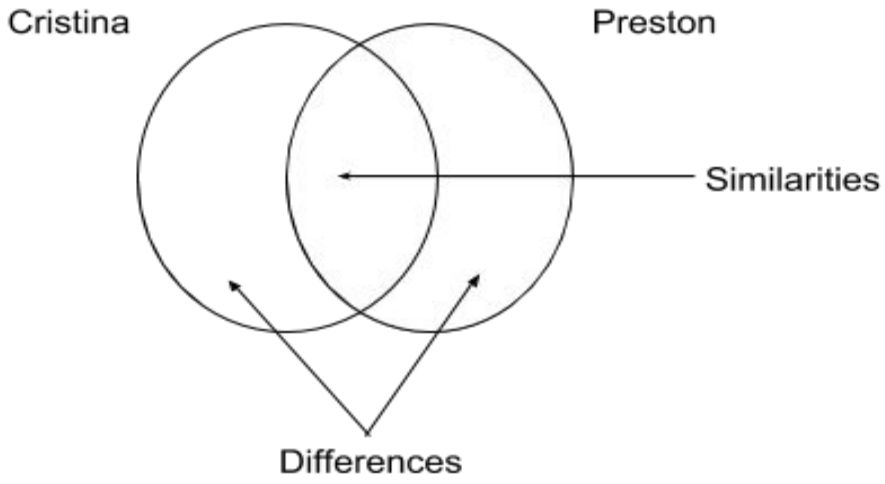


Figure 1 : Collaborative Filtering

2.1.2. Content Based Filtering

In this model, recommendations are given based on user's behaviour. User's browsing information is taken into account while recommendations are made. For example, if user visits the videos in Comedy category more, it is more likely that he or she would watch the next video under comedy genre. Content based filtering recommends similar content to the user for which he has expressed interest in the past. For example in the above example of collaborative filtering, it is clear that Preston has interest in watching videos in sci-fi category. So Content based filtering would recommend him similar movies/videos. The recommendations are based on behaviour of the user under consideration and is independent of behaviour of other users of the system.

2.1.3. Hybrid Approaches

The hybrid approach combines the aforementioned approaches to increase the efficiency of a recommender system. Hybrid approach also has the potential to make recommender system more accurate. The collaborative filtering and content based filtering face the challenge of cold start. Hybrid approach can address this to some extent. In case of hybrid approach, the system starts with content based filtering and gradually switches the focus towards collaborative filtering as the database for user information matures.

2.2. Basic Algorithms for Recommender Systems

2.2.1. Memory-based Algorithms

These algorithms try to user that is similar to active user. This algorithm uses the preferences by similar users in order to recommend something to active user.[2] In order to find the similarity between two users we need to find their correlation.

2.2.1.1. Pearson Correlation

Pearson Correlation coefficient gives an idea about how two entities are correlated with each other. This algorithm measures the linear dependence between two variables (or users) as a function of their attributes [1]. However, this dependence is not calculated on the entire dataset. Instead, it is calculated on sub-groups or neighbourhoods of dataset which are similar to each other on higher level. For example, group of users who have interest in comedy genre videos.

The pearson correlation coefficient is calculated as:

$$r = \frac{n(\Sigma xy) - (\Sigma x)(\Sigma y)}{\sqrt{[\, n\Sigma x^2 - (\Sigma x)^2 \,][\, n\Sigma y^2 - (\Sigma y)^2 \,]}}$$

Figure 2 : Pearson Correlation Coefficient [3]

The following graphs explain as what does the positive, no or negative correlation means.

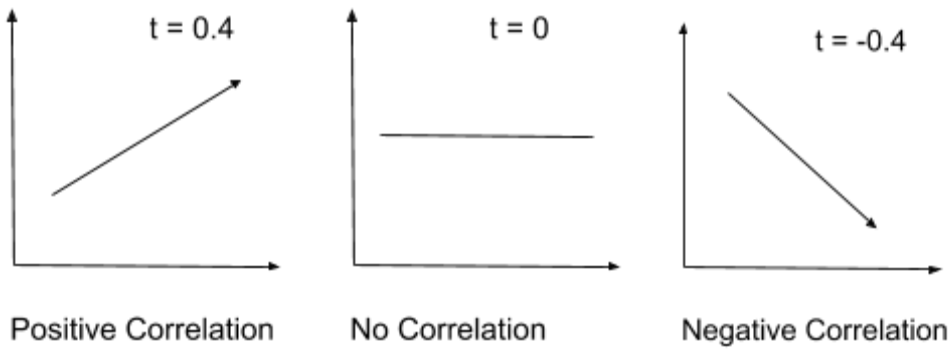


Figure 3 : Pearson Correlation

2.2.1.2. Predicting Ratings

If we want to predict the rating that the active user will give to an entity, we can do so by taking into consideration the correlation coefficient values. With positive correlation, is is more likely that the active user would agree with predicted ratings. In case of predicting ratings all weights between active user and all other users are calculated. By taking into consideration all non-negative and non-zero weights, it is asked to each one of the other users of what they think the active user would give rating to the movie. Depending on the weights, the correlation coefficient of active user and entity under consideration, the rating is predicted.

2.2.2. Model-based Algorithms

Model based algorithms use dataset of ratings in order to present recommendations. For building a model, a part of dataset is extracted, using the dataset, the model is built and using the model recommendations are made. Using this approach, we eliminate the need to bring the entire dataset in memory to do the computations. Thus, this model is beneficial with respect to speed and scalability.

2.2.2.1. Clustering Algorithms

Clustering algorithms can find a structure in a seemingly random data. Clustering is a form of unsupervised learning. These algorithms depend on finding similarity in the dataset over a feature space. The feature space can contain one or many attributes. For example, people who like comedy movies can be a feature and all people who satisfy this constraint may belong to one cluster.The number of features in a feature space determines the dimensionality of the the feature space.

The typical clustering algorithm is k-means clustering. In this the items are divided into k clusters. Initially the items are placed into random clusters. Then for each cluster, centroid is calculated. Then distance of each item from the centroid is measured. If an item is nearer to other cluster, it is moved into that cluster. After some iterations the algorithm may stabilize. i.e. no item movement is done in that iteration. This is when the algorithm terminates.

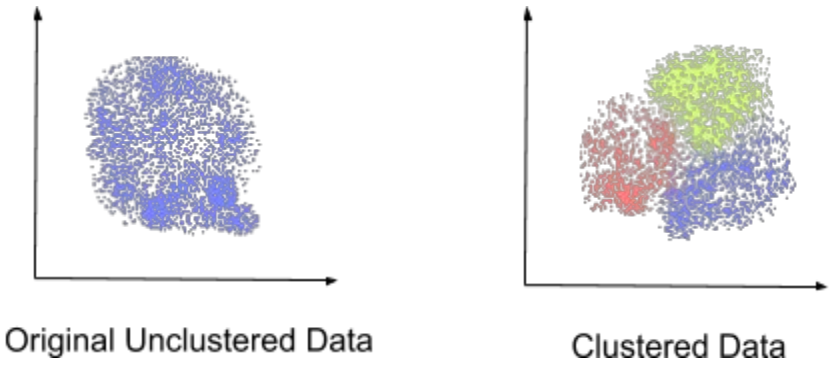


Figure 4 : K-Means Clustering [4]

2.3. Learning to Rank

2.3.1. Learning to Rank Process

Most of the existing recommender systems are either based on Collaborative filtering or content based filtering.[5] Some of them follow hybrid approach, which combines advantages of both the basic approaches and gives better performance. Generally, recommender systems are based on ratings. In these systems the recommendation problem comes to task of rating prediction. The ultimate goal of a recommender system is to generate list of recommendations. However, the intermediate step for the system is to predict the ratings. Thus, learning to rank is a supervised(or semi-supervised) application of machine learning. Training data consists of list of items or entities with a specified partial ordering between them. The order is given by making use of some sort of scores, labels given to each item in the list. The purpose of ranking model would be to rank the new, unseen data and produce a

ranked list of new data. This new list should be in some way similar to the ordering in the training data [4].

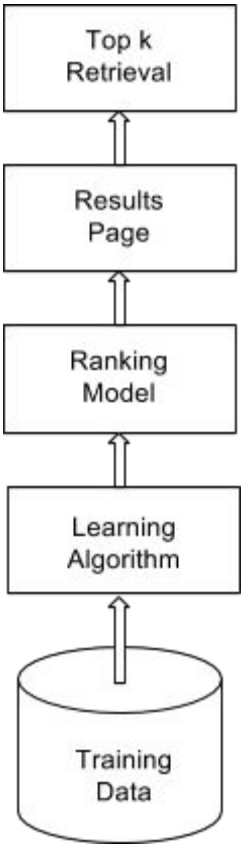


Figure 5 : Learning to Rank [4]

2.3.2. Feature Vectors

Many machine learning algorithms do require the query-document pairs to be represented by numerical vector which is called as bag of features or feature vector. The components of this vector are called as features, factors or ranking signals. These features can be divided into following groups:

a. Query Independent or Static Features: These features depend only on the document not on the query. For example: PageRank or Document length. Query independent features can be computed statically.

b. Query Dependent or Dynamic Features: These features depend on both content of the document as well as the query. Example of query dependent feature is TF-IDF score.

c. Query Level or Query Features: These feature only depend on the query not on the document. Example is number of words in the query.

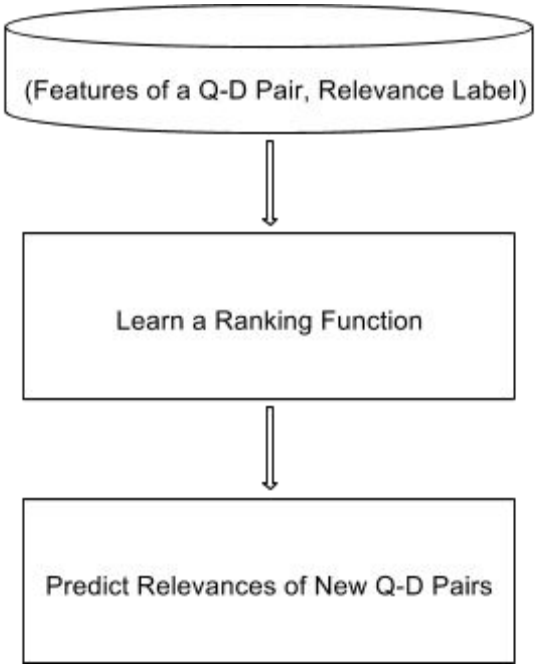


Figure 6: Feature Vector and Learning to Rank

2.3.3. Learning to Rank Approaches

Learning to rank algorithms and problems can be divided into following approaches depending on the nature of loss function and input representation.

a. Pointwise Approach: In this approach it is assumed that each query-document pair that belongs to training data has a numerical score given to it [4]. In this case, the problem of ranking comes to given a query-document pair, predict its score. It is also called as regression. For using regression methodology, a number of supervised machine learning algorithms can be used. Existing methods of classification, ordinal classification or regression can be applied [6].
b. Pairwise Approach: In this approach, ranking is transformed into pairwise classification or pairwise regression. This involves learning a binary classifier which determines which query-document pair is better from a list of query-documents.
c. Listwise Approach: The algorithms which belong to this approach try to optimize the value of above two approaches. The average value over all the query data is generally taken. Listwise approach of learning to rank is explained step by step as below.
    1. The input consists of list of queries q1 to qn. Each query qi is associated with list of documents d1 to dn. Also, each list of documents is associated with list of judgement scores or labels l1 to ln. For example l3 denotes the judgement score for document d3 with respect to query q. The judgement score can be assigned manually. The score, l3, denotes that how relevant the document d3 is with respect to query q.
    2. Each query document pair is then assigned with a feature vector. For each query q, we have associated document list di (i=1 to i=n). For each

query-document pair, we now create a feature vector. Hence each query-document pair now has a score and a feature vector associated to it. List of all feature vectors and list of scores compose an ‘instance’.[7]

3. A ranking function is created as the part of next step. The ranking function, f, generates a score corresponding to a document d. The feature vector for document d is used as an input to the ranking function. We now have a trained ranking function.The objective of learning can be stated as minimization of the total losses with respect to the training data.
4. When a new query is issued, the list of documents associated with it is determined along with their feature vectors. We use these feature vectors and our trained ranking function to generate document scores. We then sort the documents with respect to their scores and return the top ‘k’ documents as the result.[7]

This project uses listwise approach of learning to rank, the details of which are described in subsequent sections.

3. PROJECT DESIGN

3.1. Definition

3.1.1. Problem Formulation

Entity recommendation can be defined as given an entity under consideration, present/recommend similar entities. Given the large number of related entities in the knowledge base, we need to select the most relevant ones to show based on the current query of the user. Entity ranking helps to retrieve the entities, which are most relevant, based on popularity, authority, relevance etc. Following data states about input, output, data, entity and problem that this project addresses.

Input: An Entity

Output: Recommended entities given the input entity

Data: semi-structured or unstructured data

Entity: represent an object, structured data

Problem is to provide a solution to build a recommendation engine using Big Data handling technology, Apache Spark. The goal of an entity recommendation system over big data is to design a system that is scalable, efficient, and provides the best possible results for a large variety of queries.

Challenges that an entity recommendation system over big data faces are stated as below.

1. Unstructured Data: Entity recommendation over big data involves processing and storing the vast amount of unstructured data.

2. Entity Resolution and Entity Disambiguation: “Brad pitt” may refer to the actor entity “Brad Pitt” or the boxer entity “Brad Pitt (boxer)”. Moreover, there may be cases with a common meaning for the string (e.g., the entity “XXX (movie)” is not the most likely intent for query string “xxx”). Hence, the problem here is to identify the most likely intent for a given entity string.
3. Ranking: For a given entity, not all results might interest the user. We need to rank the results. There are many ranking mechanisms based on various features such as click frequency, pagerank etc.

3.1.2. Terminology

The following terms are widely used in the report:

- *Entity*: a concept or abstract that has a complete meaning by itself. In this project, entity represents an object with unique id and properties. Entity may include, but not limited to persons, subjects, records, concepts…
- *Similarity*: denotes the relevancy between an entity and a query, as a numerical value computed by a similarity functions. The higher the value, the more closely an entity relates to the query.
- *Popularity*: the concept that measures the credential of the entity, how popular is a particular entity compared to the common ground of all other entities.
- *Field/Property*: an attribute of an entity.

3.2. Technology

3.2.1. Apache Spark

Apache spark is an Open-Source data analytics cluster computing framework [8]. Spark belongs to Hadoop Open Source community. It is built on top of Hadoop Distributed File System. Although such similarities exist, Spark performs better than Hadoop in case certain specific applications. Spark is not restricted by two stage Mapreduce paradigm. It delivers 100 times better performance than Hadoop for certain applications. Spark provides the capabilities for in-memory cluster computing which allows user programs to load data into cluster's memory.

The data loaded into main memory can be used repeatedly by subsequent database accesses and it speeds up the entire response time. This property makes Spark well suited for Machine Learning algorithms.

3.2.1.1. Spark Specific Applications

Hadoop users find mapreduce programming model as deficient in two main types of jobs/applications. [9]

Iterative Jobs: Many machine-learning algorithms follow an iterative model, wherein a function is applied repeatedly on same dataset. Each iteration can be expressed as Mapreduce job. But, in case of mapreduce, each iteration/job must reload the data from disk. This reduces the performance significantly.

Interactive Analytics: Hadoop is often used to run queries on large datasets using Hive and Pig. In such cases, user should be able to load the data at once in main

memory and query it over and over again. In case of Hadoop, each query is executed as a separate job. Each job will again access the disk for loading data in memory, slowing down the entire system. To achieve these goals, Spark introduces an abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time [9].

3.2.1.2. Spark Programming Model

A 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. The elements of an RDD need not exist in physical storage; instead, a handle to an RDD contains enough information to compute the RDD starting from data in reliable storage. This means that RDDs can always be reconstructed if nodes fail [9]. As elements of the RDD need not exist in physical storage, and only a handle is enough for reconstructing it, the iterative and or interactive jobs will execute faster than that in Hadoop. In Spark, each RDD is represented by a Scala object. Spark lets programmers construct RDDs in four ways: [8]

File: From the shared file system, for example Hadoop Distributed File System

By parallelizing collection like an array: By dividing the array into number of slices. These slices can be sent to multiple nodes.

By transforming an existing RDD: A dataset with elements of type A can be transformed into a dataset with elements of type B using an operation called flatMap, which passes each element through a user-provided function of type A => List [B].

By Changing the Persistence of an existing RDD: The RDDs are by default lazy. The RDDs are not immediately reflected onto the disk. They are stored into main memory as long as they can be stored. Unless driven by any need from the user or application, the RDDs are not written back to the disk. Although, user can change the persistence properties of an RDD and change it to cache action and save action.

The **Cache Action** will mark the dataset that it will be referred to in future and should be kept in memory. The dataset marked with cache action will not be immediately written onto the filesystem such as HDFS but it would be kept in main memory. It indicates the immediate future reference.

The cache action is just a hint that the data can be used in immediate future. But if there is not enough memory on the cluster to keep the data marked cache, Spark will recompute the data as and when it will be referenced.

The **Save Action** will not leave the dataset lazy. It will save the dataset and write onto the filesystem. The saved version of the file will be referred to in future operations.

The concepts of save and cache actions is fairly similar to virtual memory and it is used to impose the fault tolerance in Spark.

3.2.1.3. Parallel Operations

Parallel operations need to be performed for scalability. Following parallel operations are possible in case of resilient distributed datasets-

Reduce: Reduce operation is fairly similar to reduce in Hadoop. Reduce will combine the dataset elements to produce the result. The dataset elements will be combined associatively to produce a cumulative result.

Collect: The user program will be able to get the results from all the nodes using collect operation. Collect operation will send the all the elements of the dataset to the driver or user program. If we consider of processing an array, user can update the array in parallel. User can parallelize the array, map the array to the different nodes and collect the results. The collect operation here will give the user the independence to collect the results at once.

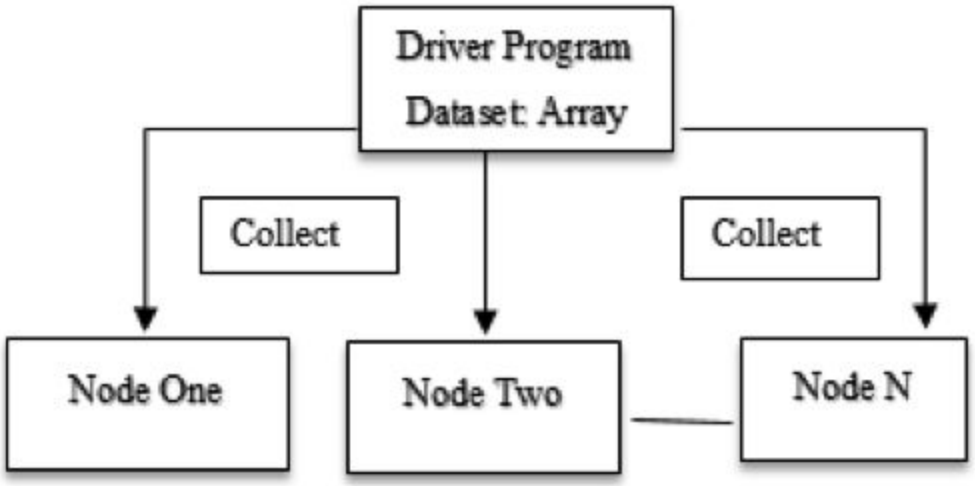


Figure 7: Spark Collect Operation

As shown in figure 6, the dataset elements will be mapped, parallelized and processed, and collected back by the driver program. In case of Spark, that data will be collected by one single thread. However, according to Spark documentation, the one thread reducer is enough for implementation of multiple algorithms that Spark aims for.

For-Each: For-each passes the elements via the functions provided by users. This is used for some repetitive kind of functions which will collect the data and produce some kind of cumulative result by processing the collected data.

3.2.1.4. Shared Variables

In Spark, the operations like map, reduce, filter are invoked by users or programmers. Users generally invoke these functions by passing functions to Spark. Hence the variables that these functions use should be within the scope where these functions will get executed. If a worker node is supposed to execute a function, then the variables needed by that function should be copied onto worked node. Spark framework does exactly the same. Although this is what happens generally in case of Spark, it also lets users choose two special types of variables. These variables are named as **Broadcast Variables** and **Accumulators** .

Broadcast variables: As explained above, with function, the required variables are copied onto the worker node where that function is being executed. But there might be a use-case wherein many of the functions might be using a single data variable such as lookup table. In this case, it is not performance beneficial if we copy the look up table onto each of the nodes over and over again. Instead, make the common data as broadcast variable. By doing so, programmer can rest assured that the variable value is only copied to each worker once.

Accumulators: Accumulators are like counters. They have add operation and zero value. They can be added to by different nodes. They are fault tolerant due to their

add-only property. They can be used in case of parallel sums. Map-Reduce paradigm also uses them.

4. IMPLEMENTATION

This section describes the of implementation details of the recommendation engine. This section contains the code snippets and relevant description of implementation of the project.

4.1. Knowledge Base Creation

Recommendation engine takes a large entity graph as input, and applies a ranking function to extract a weighted subgraph consisting of the most important entities, their most important related entities, and their respective types.

4.2. Knowledge Base Acquisition

For knowledge acquisition, we utilize DBpedia dataset as well as Wikipedia clickstream dataset.

Why Dbpedia?

Wikipedia articles consist mostly of free text, but also include structured information embedded in the articles, such as “infobox” tables, categorization information, images and links to external web pages. Dbpedia project aims to extract this structured content from Wikipedia articles and represent in Resource Description Format (RDF), allowing users to semantically query relationships and properties associated with Wikipedia

resources, including links to other related datasets. By using Dbpedia datasets (instead of raw Wikipedia data), we can simplify the knowledge acquisition phase.

Dbpedia datasets are available in N-triples format, which is a line-based plain text serialization format for Resource Description Framework (RDF) graph. In this format, each line (or a statement) consists of three parts separated by one or more whitespace characters and ending with a '.' Character. E.g.

```
<http://dbpedia.org/resource/Aristotle> <http://xmlns.com/foaf/0.1/name> "Aristotle"@en .
```

Figure 8: DBpedia Data Example

Here,

- <http://dbpedia.org/resource/Aristotle> is a **subject**
- <http://xmlns.com/foaf/0.1/name> is a **predicate**
- "Aristotle"@en is an **object**

We utilize following three Dbpedia datasets to build an entity graph,

- Resource properties data, which provides RDF representation for various resources and their properties.
- Mapping between Dbpedia resource and corresponding Wikipedia page id. This information is important since the Wikipedia clickstream dataset (explained below) refers to Wikipedia article in terms of its page id. Also the Apache Spark (Graphx) module used for knowledge extraction (explained below) requires a unique identifier for every vertex present in the graph. By using a well-known id for every resource, we simplify its representation and processing.

- Relationships between DBpedia resources constructed using the associated page links between Wikipedia articles.

Why Wikipedia clickstream?

Wikipedia clickstream dataset provides counts of (referer, resource) pairs extracted from request logs of Wikipedia. A referer is an HTTP header field that identifies the address of the webpage that is linked to the resource being requested. The data shows how people get to a Wikipedia article and what links they click on. In other words, it gives a weighted network of articles, where each edge weight corresponds to how often people navigate from one page to another.

Since DBpedia datasets provide relationships between entities based on **content**, the corresponding entity graph is relatively static. On the other hand, clickstream dataset provides relationships between entities based on **current trends**, hence is relatively dynamic. **e.g.** based on Feb2015 dataset, starting from page 'Leonardo_Dicaprio' users accessed 'The_Revenant_(2015_film)" page more frequently as compared to "The_Titanic_(1997_film)" (8013 vs. 2974).

Also clickstream dataset is very useful for calculating popularity features e.g. which movies are currently trending (based on Wikipedia clickstream logs).

Please refer to following URLs for the datasets mentioned above.

- http://data.dws.informatik.uni-mannheim.de/dbpedia/2014/en/instance_types_en.nt.bz2
- http://data.dws.informatik.uni-mannheim.de/dbpedia/2014/en/page_links_en.nt.bz2
- http://data.dws.informatik.uni-mannheim.de/dbpedia/2014/en/page_ids_en.nt.bz2
- http://files.figshare.com/1905609/2015_01_clickstream.tsv.gz

Database Tested on: Wikipedia Database (DBPedia)

4.3. Knowledge Base Construction and Entity Graph Construction

For constructing knowledge base, we use Apache Spark cluster computing framework. The fundamental programming abstraction is Resilient Distributed Datasets (RDD), which provides a logical view for the collection of data partitioned across multiple machines. RDDs can be created by referencing datasets in external storage systems, or by applying coarse-grained transformations (e.g. map, filter, reduce, join) on existing RDDs.

For graph computations, we utilize Graphx APIs provided by Apache Spark. GraphX unifies ETL, exploratory analysis, and iterative graph computation within a single system. We can view the same data as both graphs and collections, transform and join graphs with RDDs efficiently, and write custom iterative graph algorithms using the Pregel API.

As part of knowledge base construction, we have defined RDDs representing vertices as well as edges of the Entity Graph. During this phase, we preprocess the raw data provided by the DBpedia data-set to generate an optimal representation expected by the Graphx APIs (in order to avoid costly RDD joins during run-time).

During the **preprocessing phase**, following steps are required to build a DBpedia entity RDD,

- Parse the DBpedia *instance_types_en.nt* file to prepare a Scala tuple of type (String, List [(String, String)] for every line in the input file. This tuple represents a resource label (first parameter) and a list of properties (or key/value pairs).
- In the second phase we combine all properties for a given resource.

```
val nodes: RDD [(String, List [(String, String)])]   = sc.textFile("instance_types_en.nt").
                                                        map (simple_nt2kv).
                                                        reduceByKey (fold_dict)
```

Figure 9: Code Snippet- Combine Resource Properties

Next, we build RDD representing mapping between DBpedia resource URL and the Wikipedia page id.

```
val pageIds: RDD [(String, Long)]  = sc.textFile ("page_ids_en.nt").
                                     filter (elem => !elem.startsWith ("#")).
                                     map (parse_pageid)
```

Figure 10: Code Snippet- Mapping between DBpedia Resource and Wikipedia ID

Finally we build GraphX vertex RDD by joining the nodes RDD and pageIds RDD created above (so as to use the Wikipedia page id as the vertex id in the Graph computations).

```
val vertexRDD: RDD [(Long, WikiNode)]  = nodes.join (pageIds).
                                          map (x => (x._2._2, WikiNode (x._1, x._2._1)))
```

Figure 11: Code Snippet- GraphX Vertex RDD

For building GraphX edge RDD, we need to parse the page_links_en.nt file. For every line in this file, we get a Scala tuple (String, String, String) representing URLs of two DBpedia resources and the corresponding relationship name. Since GraphX API require an edge to be represented by ids of two vertices participating in this edge (optionally with edge attributes). This is achieved by transforming (String, String, String) tuple to (Long, String, Long) tuple using multiple joins.

```
val pageLinks: RDD[(Long,String,Long)]                = sc.textFilepage_links_en.nt").
                                        map(parse_pagelink).
                                        map({case (k1,k2,k3) => (k1, (k2,k3)) }).
                                        join(dbpediaPageIds).
                                        map({ case (_, ((k2,k3),v1)) => (k3, (v1,k2)) }).
                                        join(dbpediaPageIds).
                                        map({ case (_, ((v1,k2),v2)) => (v1,k2,v2) })
```

Figure 12: Code Snippet- GraphX Edge RDD

Final step is to build the graph object using the vertex and edge RDDs created above.

```
val graph = Graph(vertexRDD, pageLinks)
```

Figure 13: Code Snippet- Building Graph Object

Using the entity graph built in above steps, we can extract the features as popularity features and graph-theoretic features.

4.4. Feature Extraction

4.4.1. Popularity Features

We have used Wikipedia click-stream data-set for this purpose. This data shows how people get to a Wikipedia article and what articles they click on next. In other words, it gives a weighted network of articles, where each edge weight corresponds to how often people navigate from one page to another.

For the recommendation engine, we are interested in finding about the navigation between Wikipedia articles only. i.e. given a Wikipedia article A, find out how many times A is referred by another Wikipedia article B. We normalize this count by converting it to a percentage value (by summing referral counts for all the Wikipedia articles linking to A). This calculation is carried out using Apache Spark processing framework. Here is the relevant code-snippet for this functionality.

```
def filterLogLine(ln: String): Boolean = ln.split("\t", -1) match {
  case Array(_,_,_,_,_,typeStr) => ("link".equals(typeStr))
}

def parseLogLine(ln: String): (Long, (Long, Long)) = ln.split("\t", -1) match {
  case Array(p_id,c_id,n,prev_title,curr_title,typeStr) => (c_id.toLong, (p_id.toLong, n.toLong))
}

def avg(elems: Iterable[(Long, Long)]): Iterable[(Long, Double)] = {
  val count: Double  = elems.map(x => x._2).reduceLeft(_ + _)
  elems.map(x => (x._1, x._2/count))
}

val stream: RDD[(Long, Iterable[(Long, Double)])]= sc.textFile("2015_02_clickstream_filtered.tsv").
                                                    filter (filterLogLine).
                                                    map (x => parseLogLine(x)).
                                                    groupByKey().
                                                    map(x => (x._1, avg(x._2)))
```

Figure 14: Code Snippet- Calculating Reference Count

4.4.2. Graph-Theoretic Features

We use Pagerank value as one of the graph-theoretic features. The pagerank value can be computed by applying PageRank algorithm implementation provided in the Spark GraphX module on the entity graph constructed above. Here is the relevant code-snippet for this functionality.

```
//The entity graph created using the DBpedia article as the vertex and the //hyperlink
between two articles as the relationship.
val graph = Graph(vertexRDD, pageLinks)
// 0.01 parameter value describes the tolerance allowed at convergence
val page_ranks RDD[(Long, Double)]= graph.pageRank(0.01).vertices
```

Figure 15: Code Snippet- Calculating PageRank

Please note that calculating page_rank for the entire DBpedia collection is extremely resource-intensive process requiring large compute cluster. Hence for prototyping purpose, we have used pre-calculated PageRank values for the DBpedia collection from following source,

http://people.aifb.kit.edu/ath/

4.5. Preparing a Feature Vector

For ranking purpose we need to build a vector of all features for a given DBpedia article. This is implemented by joining the RDDs corresponding to individual features together. Here is the relevant code-snippet for this functionality.

```
/**
 * This class represent a Feature vector for a given DBpedia article (A). The click_logs
 * parameter defines mapping between DBpedia article_id (B) and the relevant percentage
 * score for number of referrals from the article B to A.
 **/
case class Features(pageRank : Double, click_logs: Iterable[(Long, Double)])

/**
 * A RDD representing click_log feature values where each tuple in the RDD represents a
 * DBpedia article_id (A) as the first parameter and mapping between DBpedia article_id (B)
 * and the relevant percentage score for number of referrals from the article B to A as the
 * second parameter.
 **/
val clickStream: RDD[ (Long, Iterable[(Long, Double)])] = …

/**
 * A RDD representing PageRank values where each tuple in the RDD represents a
 * DBpedia article_id (A) as the first parameter and PageRank value as the second
 * parameter.
**/
val pageRank: RDD[(Long, Double)] = …

/**
 * A RDD representing feature vector for DBpedia collection
 **/
val features: RDD[ (Long, Features)] =
        pageRank.join(clickStream).
                map({case (v_id, (p_rank, l)) => (v_id, Features(p_rank, l))})
```

Figure 16: Code Snippet- Preparing Feature Vector

4.6. Ranking

4.6.1. ListNet Training Algorithm

Following is the pseudo code for listnet training algorithm.

1. Input: training data consists of
    a. Set of queries Q = {q(i)}, i=1,2,..,m

i. List of documents d(i) = $\{d^{(i)}_j\}$

1. List of judgments (scores) $y^{(i)} = \{y^{(i)}_j\}$
2. Feature vector $x^{(i)}_j$ for each query-document pair

2. Parameter: number of iterations T and learning rate η
3. Initialize parameter ω (*A one-dimensional vector of size equal to number of features*) (Double value between 0.0 to 1.0)
4. For t = 1 to T

   do

   For i = 1 to m

   do

   Input $x^{(i)}$ of query $q^{(i)}$ to Neural Network and compute score list $z^{(i)}(f\omega)$ with current ω

   Compute gradient $\Delta\omega$

   Update $\omega = \omega - \eta * \Delta\omega$

   done

   done
5. Output: Neural Network model ω [10]

4.6.2. Preparing Training Data-set

As part of the training phase, I prepared a set of queries and relevant labeled results. Here, each query is a regular expression identifying a specific DBpedia entity e.g. Kate_Winslet. The query result corresponds to all entities connected to the specified entity (s) ranked with respect to their relevance. We also provide gold relevance score

as a “label” for model building using ListNet algorithm. The label in this case is an integer value between 1 to 10 (10 being most relevant and 1 being least relevant).

Please refer to following code snippet to understand how we extract the “connected” entities and their corresponding feature vectors.

```
/**
 * This class represents a set of features computed for a specific relationship between two DBpedia
 * vertices.
 **/
case class FeatureVector (page_rank: Double, click_log: Double)

/**
 * This function returns a feature vector for a specified DBpedia entities.
 * @param src_id An unique id for a DBpedia entity which is the “source” for this relationship
 * @param dst_id An unique id for a DBpedia entity which is the “destination” for this relationship
 * @param features Cumulative features for the DBpedia entity specified by <code>dst_id</code>
 * @return  feature vector for a specified DBpedia entities.
 **/
def compute_feature_vector(src_id: Long, dst_id: Long, features: WikiNodeFeatures): FeatureVector = {
   val click_log = features.click_logs.find(x => x._1 == src_id) match {
     case Some(y) => y._2
     case None    => 0d
   }
   FeatureVector(features.pageRank, click_log)
 }
```

Figure 17: Code Snippet- Extract Connected Entities and Feature Vectors

```
/**
* This function computes the "connected" entities for one or more entities representing the
* specified <code>regex</code> along with their feature vectors (to be used for calculating the
* relevance score).
* @param vertices An RDD representing the vertices of the DBpedia graph
* @param pageLinks An RDD representing the edges of the DBpedia graph
* @param features An RDD representing the pre-computed features for the DBpedia entities
* @param filterExpr A filter function which can (optionally) filter the vertices to be considered for
*                   ranking
* @param regex A regular expression used to identify the DBpedia entities used to perform
*              graph search.
* @return A RDD representing information about the "connected" entities and their feature vectors.
**/
def compute_features( vertices: RDD[(Long, WikiNode)], pageLinks: RDD[(Long, Long)],
    features: RDD[(Long, WikiNodeFeatures)], filterExpr: (WikiNode => Boolean), regex: String): RDD
[(Long, (Long, WikiNode), FeatureVector)] = {
  vertices.filter(x => x._2.title.matches(regex)).
        join(pageLinks).
        map({ case (src_id, (v_info, dst_id)) => (dst_id, src_id) }).
        join(vertices).
        map({ case (dst_id, (src_id, dst_v_info)) => (dst_id, (src_id, dst_v_info)) }).
        filter(x => filterExpr(x._2._2)).
        join(features).
        map({ case (dst_id, ((src_id, dst_v_info), features)) => {
            (src_id, (dst_id, dst_v_info), compute_feature_vector(src_id, dst_id, features))
          }})}
```

Figure 18: Code Snippet- Compute Features

Using this implementation, I prepared (and saved) the Spark RDD representing the result of the query along with the relevant feature vectors. After this, for each query, I inspected the results and manually assigned "gold relevance" labels. e.g. for query "Big_Bang_Theory", here is a snippet of training data-set:

```
#entity_name, page_rank_score, click_log_percentage, label
wiki:David_Saltzberg,0.18463,0.94608,10
wiki:Faye_Oshima_Belyeu,0.26536,0.93448,10
wiki:Steven_Molaro,0.30732,0.82505,10
wiki:Kevin_Sussman,0.47299,0.74140,9
wiki:Melissa_Rauch,0.30688,0.70938,9
wiki:Bill_Prady,0.50519,0.60649,9
wiki:Simon_Helberg,0.59944,0.58539,9
...
```

Figure 19: Training Data Set

4.6.3. ListNet Ranking Algorithm

Please refer to following snippet for the ListNet algorithm using Apache Spark framework. This function accepts a training-set along with following parameters,

- iterations (Number of iterations to be carried out during training phase)
- step_size (the learning rate)

and returns the training-model which can be used to predict the relevance score for a given DBpedia entity link.

```
/**

case class DocData(doc: String, features: Array[Double], label: Int)
case class Instance(query: String, docs: List[DocData])

def score(features: Array[Double], weights: ArrayBuffer[Double]): Double = {
   require(features.size == weights.size)
   var result = 0d
   for (i <- 0 until features.size) {
     result += features(i) * weights(i)
   }
   result
}

def initializeWeights(dimensions: Int): ArrayBuffer[Double] = {
  val r = scala.util.Random
  val result = ArrayBuffer.fill(dimensions)(0d)
  for (i <- 0 until dimensions) {
    result(i) = r.nextDouble()
  }
  result
}
```

Figure 20: ListNet Implementation - 1

Following figure depicts the code snippet for listnet ranking function

```
def listNet(sc: SparkContext, trainingSet: List[Instance], iterations: Int, stepSize: Double ): ArrayBuffer[Double] =
{
  val weights = initializeWeights(2)
  for (i <- 1 to iterations) {
   val gradient = sc.accumulator(ArrayBuffer.fill(2)(0d))(ArrayAccumulatorParam)
   val loss = sc.accumulator(0.0)
   for (q <- trainingSet) {
    val expRelScores = q.docs.map(y => math.exp(y.label.toDouble));
    val sumExpRelScores = expRelScores.reduce(_ + _)
    val P_y = expRelScores.map(y => y / sumExpRelScores);

    val ourScores = q.docs.map(y => score(y.features, weights))
    val expOurScores = ourScores.map(z => math.exp(z))
    val sumExpOurScores = expOurScores.reduce(_ + _)
    val P_z = expOurScores.map(z => z / sumExpOurScores)

    var lossForAQuery = 0.0;
    var gradientForAQuery = ArrayBuffer.fill(2)(0d)
    for (j <- 0 to q.docs.length - 1) {
     val t = q.docs(j).features.map(x => x * (P_z(j) - P_y(j)))

     ArrayAccumulatorParam.addInPlace(gradientForAQuery, t)
     lossForAQuery += -P_y(j) * math.log(P_z(j))
    }
    gradient += gradientForAQuery
    loss += lossForAQuery
   }
   ArrayAccumulatorParam.subtractInPlace(weights, gradient.value.map(y => y * stepSize))
  }
```

Figure 21: ListNet Implementation - 2

The pseudo code for ListNet Ranking algorithm is as follows

1. Input: A regular expression corresponding to the DBpedia entity to be searched (e.g. Jennifer Aniston)
2. Figure out all DBpedia entities matching provided regular expression (denoted by X)
3. Figure out all DBpedia entities referred by the entities identified during step (2).
4. Compute the feature vectors for each entity identified during step (3)

5. Apply the ListNet ranking function on each feature vector calculated in step (4) to calculate document scores
6. Sort the entities identified in step (3) with respect to their scores identified in step (4) in decreasing order and return top-K entities.

Please refer to following code snippet for ranking algorithm based on ListNet

```
/**
* This function computes the "connected" entities for one or more entities representing the
* specified <code>regex</code> in a ranked order.
*
* @param vertices An RDD representing the vertices of the DBpedia graph
* @param pageLinks An RDD representing the edges of the DBpedia graph
* @param features An RDD representing the pre-computed features for the DBpedia entities
* @param filterExpr A filter function which can (optionally) filter the vertices to be considered for
*                   ranking
* @param regex A regular expression used to identify the DBpedia entities used to perform
*              graph search.
* @param weights ListNet training model
* @param k Number of results to be returned to the caller (top-k)
* @return An array representing information about the "connected" entities in a ranked order.
**/
def rank(vertices: RDD[(Long, WikiNode)],
         pageLinks: RDD[(Long, Long)],
         features: RDD[(Long, WikiNodeFeatures)],
         filterExpr: (WikiNode => Boolean),
         regex: String,
         weights: ArrayBuffer[Double],
         k: Int): Array[(Long, WikiNode)] = {
   compute_features(vertices, pageLinks, features, filterExpr, regex).
        map({case (src_id, (dst_id, dst_info), f_v) =>
          (dst_id, dst_info, score(Array(f_v.page_rank, f_v.click_log), weights))}).
        sortBy(_._3, false).
        map(x => (x._1, x._2)).
        take(k)
  }
```

Figure 22: ListNet Ranking

5. PERFORMANCE

The performance of the recommendation engine is explained in the section below. We will first describe the cluster details and will also describe the input data sizes, the time it takes for various algorithms to finish.

5.1. Cluster Details

- OS : CentOS 6.5
    - 1 host : 8 CPU cores + 15 GB RAM
    - 4 hosts : 4 CPU cores + 7 GB RAM
- 1 Spark master and 4 Spark worker nodes
- HDFS installed on all nodes (replication level = 3)
- Apache Spark version 1.5.0
- Apache HDFS version 2.6
- Apache Zookeeper version 3.5

5.2. Input Data Sizes

Description	Input Data Type	Size on Disk	Number of Elements
Entity Attributes File	instance_types_en.nt	4.17 GB	28031876
DBPedia Page Ids File	page_ids_en.nt	2.04 GB	13494821
DBPedia Page Links File	page_links_en.nt	23.37 GB	152913360
PageRank Values File	pagerank_en_2014.tsv	1.31 GB	19540318

ClickStream Values File	2015_02_clickstream.tsv	905.75 MB	14419586
DBpedia Graph Vertices	Java Objects	1.44 GB	3022345
DBpedia Graph Edges	Text Format	5.1 GB	142116737
Features	Java Objects	863.79 MB	2046154

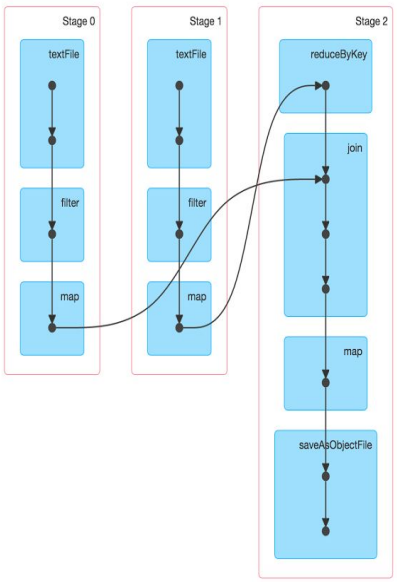
Table 2: Input Data Sizes

5.2.1. Training Data Size

Training Data consists of five queries, documents related to those five queries, with 80 records per query. In all there are 400 records which are labeled manually on a scale of 1 to 10.

5.3. Run Time Performance Details

5.3.1. Compute DBpedia Graph Vertices



Completed Stages (3)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	saveAsObjectFile at <console>:88 +details	2015/11/15 18:13:11	1.1 min	31/31		1421.1 MB	570.9 MB	
1	map at <console>:81 +details	2015/11/15 18:11:43	1.5 min	31/31	3.9 GB			247.8 MB
0	map at <console>:79 +details	2015/11/15 18:11:43	24 s	16/16				323.1 MB

Table 3: Compute DBPedia Graph Vertices

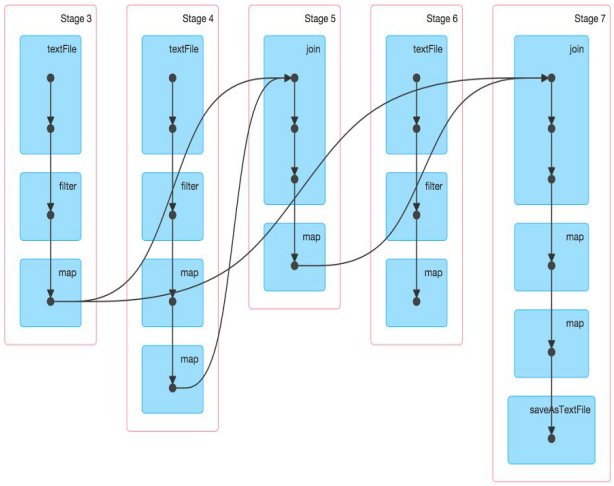
The computation of DBpedia graph vertices is carried out in three stages as follows,

- During stage 0, it reads page_ids_en.nt file containing the mapping between the Dbpedia URL (e.g. http://dbpedia.org/page/Friends) and the associated unique page identifier and outputs an RDD consisting of scala tuples of type (String, Long) representing (entity_name, entity_id). The filter operator eliminates comments in the file (i.e. all the lines starting with “#” character).
- During stage 1, it reads instance_types_en.nt file consisting of various attributes

for a given DBpedia entity. The filter operator eliminates comments in the file (i.e. all the lines starting with “#” character). The output of this stage is an RDD consisting of scala tuples of type (String, (String, String)) which represents (entity_name, (entity_attr_name, entity_attr_value)).

- During stage 2, it applies reduceByKey operator on the result of stage 1 to aggregate all the attributes for a given entity. The result of this operator is joined with the result of stage 0 to aggregate information of the entities in the form of an RDD containing scala tuples (entity_name, (entity_id, List[(attr_name, attr_val)])). The subsequent **map** operation converts the result of join operation into a tuple (Long, WikiNode) in order to simplify storing & processing of this information. Finally the result of the map operation is stored to HDFS using **saveAsObjectFile** API.

5.3.2. Compute DBpedia Graph Edges



Completed Stages (5)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
7	saveAsTextFile at <console>:84 +details	2015/11/15 18:36:16	2.9 min	175/175		5.4 GB	4.0 GB	
5	map at <console>:86 +details	2015/11/15 18:32:46	3.5 min	175/175			3.3 GB	3.7 GB
6	map at <console>:79 +details	2015/11/15 18:27:28	5.5 min	16/16	1950.0 MB			338.6 MB
4	map at <console>:84 +details	2015/11/15 18:27:28	5.3 min	175/175	21.8 GB			3.0 GB
3	map at <console>:79 +details	2015/11/15 18:27:28	19 s	16/16	1950.0 MB			338.6 MB

Table 4: Compute DBPedia Graph Edges

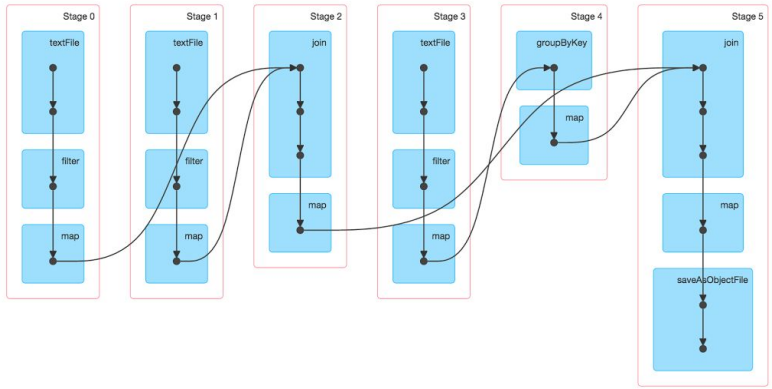
- During *stage 3*, it reads page_ids_en.nt file containing the mapping between the Dbpedia URL (e.g. http://dbpedia.org/page/Friends) and the associated unique page identifier and outputs an RDD consisting of scala tuples of type (String, Long) representing (entity_name, entity_id). The filter operator eliminates comments in the file (i.e. all the lines starting with "#" character).
- During *stage 4*, it reads page_links_en.nt file consisting of information about relationships between DBpedia entities. The filter operator eliminates comments in the file (i.e. all the lines starting with "#" character). The result of this stage is

an RDD consisting of scala tuples of type (String, (String, String)) representing (src_entity_name, (rel_type, dest_entity_name)).

- During stage 5, it **join**s the results of stage 3 & 4. The result of the join operation is transformed to a form (dest_entity_name, (src_entity_id, rel_type)) using subsequent **map** operation.
- During *stage 6*, it reads page_ids_en.nt file containing the mapping between the Dbpedia URL (e.g. http://dbpedia.org/page/Friends) and the associated unique page identifier and outputs an RDD consisting of scala tuples of type (String, Long) representing (entity_name, entity_id). The filter operator eliminates comments in the file (i.e. all the lines starting with "#" character). Note that this stage is identical to the *stage 3*.
- During stage 7, it **join**s the results of stage 3 & 5. The result of the join operation is transformed to a form (src_entity_id, rel_type, dest_entity_id) using subsequent **map** operation. Finally the result of the map operation is stored to HDFS using **saveAsTextFile** API. (Note that the second map operation converts the scala tuple (src_entity_id, rel_type, dest_entity_id) into a string for simplifying storage).

5.3.3. Compute Features for Graph Vertices

(using page_rank & click_stream data)



Completed Stages (6)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
5	saveAsObjectFile at <console>:47 +details	2015/11/15 20:06:56	18 s	16/16		350.4 MB	245.4 MB	
2	map at <console>:44 +details	2015/11/15 20:05:59	57 s	16/16			872.2 MB	85.9 MB
4	map at <console>:40 +details	2015/11/15 20:05:52	19 s	7/7			152.5 MB	159.5 MB
3	map at <console>:38 +details	2015/11/15 20:05:16	37 s	7/7	864.2 MB			152.5 MB
1	map at <console>:42 +details	2015/11/15 20:05:16	43 s	10/10	1247.4 MB			553.9 MB
0	map at <console>:40 +details	2015/11/15 20:05:16	21 s	16/16				318.3 MB

Table 5: Compute Features for Graph Vertices

- During *stage 0*, it reads page_ids_en.nt file containing the mapping between the Dbpedia URL (e.g. http://dbpedia.org/page/Friends) and the associated unique page identifier and outputs an RDD consisting of scala tuples of type (String, Long) representing (entity_name, entity_id). The filter operator eliminates comments in the file (i.e. all the lines starting with "#" character).
- During *stage 1*, it reads pagerank_en_2014.tsv file containing the mapping between the DBpedia URL (e.g. http://dbpedia.org/page/Friends) and the associated page_rank value and outputs an RDD consisting of scala tuples of

type (String, Double) representing (entity_name, page_rank).The filter operator eliminates comments in the file (i.e. all the lines starting with “#” character).

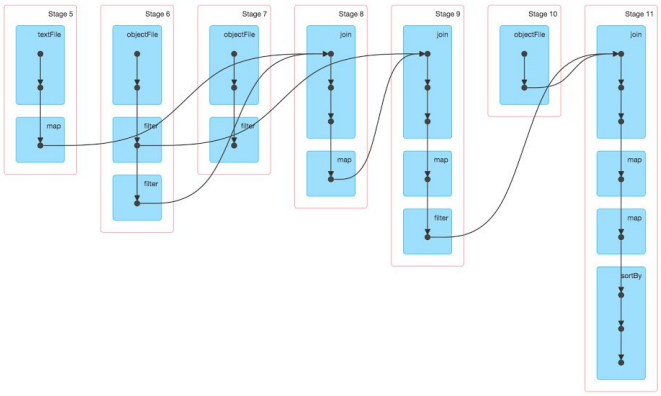
- During *stage 2*, the results of stage 0 & 1 are **join**ed (and transformed) to prepare an RDD containing tuples (Long, Double) representing (entity_id, page_rank).
- During *stage 3*, it reads 2015_02_clickstream_filtered.tsv file consisting of click-logs for the dbpedia entities. The filter operator eliminates comments in the file (i.e. all the lines starting with “#” character). The result of this stage is an RDD consisting of scala tuples of type (Long, (Long, Long)) representing (referer_entity_id, (referred_entity_id, click_count)).
- During *stage 4*, it performs **groupByKey** operation on the result of the stage 3 to aggregate the click-log details for a given referer entity in the form of (Long, List[(Long, Long)]). The subsequent **map** operation transforms the result of join operation to represent percentage of clicks from referer_entity to the referred_entities as a scala tuple (Long, (Long, Double)) representing (referer_entity_id, (referred_entity_id, percentage_clicks))
- During stage 5, the results of stage 2 & 4 are **join**ed (& transformed) together to aggregate the features for a given DBpedia entity in the form of a scala tuple (Long, WikiNodeFeatures) where WikiNodeFeatures type represents the features for a given DBpedia entity (containing page_rank as well as click_log details). Finally the result of the map operation is stored to HDFS using **saveAsObjectFile** API.

5.3.4. Compute ListNet Training

Manual Labeling: Depends on number of records .ListNet Training Algorithm time also depends on the number of records. Currently, it takes a second to come up with result.

5.3.5. Compute ListNet ranking

Ranking algorithm time depends on query under consideration and total number of records associated with the query. It is currently in the magnitude of seconds.



Completed Stages (7)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
11	sortBy at package.scala:126 +details	2015/11/29 13:36:20	3 s	175/175			138.5 MB	
9	filter at package.scala:99 +details	2015/11/29 13:36:09	11 s	175/175			221.7 MB	281.4 KB
8	map at package.scala:96 +details	2015/11/29 13:35:33	36 s	175/175			1457.7 MB	17.4 KB
10	objectFile at <console>:38 +details	2015/11/29 13:34:21	1.8 min	16/16	350.4 MB			138.2 MB
6	filter at package.scala:94 +details	2015/11/29 13:34:21	1.2 min	31/31	1421.1 MB			4.9 KB
5	map at <console>:39 +details	2015/11/29 13:34:21	1.2 min	175/175	5.4 GB			1458.3 MB
7	filter at <console>:39 +details	2015/11/29 13:34:21	1.3 min	31/31	1421.1 MB			221.7 MB

Table 6: ListNet Ranking

- In *stage 5*, it reads the DBpedia graph edges (computed above).
- In *stage 6*, it reads the DBpedia graph vertices (computed above). The final **filter** operation is used to figure out all the DBpedia entities whose title is matching the specified regular expression.
- In *stage 8*, it **join**s (and transforms) the results of stage 5 & 6 to figure out all the DBpedia vertices related to the once discovered as part of stage 6. The result of

this stage is an RDD consisting of scala tuples of the form (Long, Long) representing (dest_entity_id, src_entity_id).

- In *stage 9*, it **join**s (and transforms) the RDD representing DBpedia vertices and the result of stage 8 in the form of an RDD consisting of scala tuples of type (Long, (Long, WikiNode)) representing (dest_entity_id, (src_entity_id, src_info)). The final **filter** operation is used to figure out all the DBpedia entities to be considered for ranking purpose. The result of this stage is an RDD consisting of scala tuples of the type (Long, (Long, WikiNode)) representing (dest_entity_id, (src_entity_id, src_info)).
- In *stage 10*, it reads an RDD representing features for DBpedia entities (computed above).
- In *stage 11*, it **join**s the results of stage 9 & 10 to aggregate information about all the DBpedia entities related the specified regular expression along with their feature vectors. The subsequent **map** operations apply the ListNet ranking function on each entity and compute the score. Finally it applies **sort** operation to get the list of ranked entities in descending order.

6. CONCLUSION

This project implements a recommendation engine using Apache Spark. Results from this project indicate the advantages of using Apache Spark for distributed, big-data processing, especially for algorithms which are iterative in nature. The algorithm that the recommendation engine uses is ListNet algorithm. The recommendation engine can handle large amount of data as Apache Spark is a big-data handling technology that has in-built properties for processing iterative and interactive algorithms, like that of ListNet, faster than traditional map-reduce paradigm. Usage of Apache Spark has helped the implementation of this project as the project makes use of machine learning algorithms. Spark has a concept of distributed memory abstraction which is called as Resilient Distributed Dataset, which helps reduce disk writes and promotes the in-memory data processing. This project makes use of the distributed memory abstraction layer of Apache Spark to help implement the machine learning algorithms for recommendation engine and to help process vast amount of data in reasonable time.

This project could be further developed by making use of more number of diverse features for the recommendation process. Different Query Independent and Query Dependent features can be implemented to help the recommendation engine to be more accurate.

REFERENCES

1. Jones, Tim M. “Introduction to Approaches and Algorithms, Recommender Systems” *IBM Developerworks.* IBM Corporation, 10 January 2015
2. Lew, Daniel. Sowell, Ben. Steinberg, Leah. Tuladhar, Amrit. “Recommender Systems” *A computer Science Comprehensive Exercise*. Carleton College, Northfield, MN
3. Statistics How To. Internet. statisticshowto.com/ what-is-the-pearson-correlation-coefficient, 20 December 2013
4. Wikipedia article. Internet. “K-means Clustering” en.wikipedia.org/wiki/K-means_clustering
5. Sun, Jiankai. Wang, Shuaiqiang. Gao, Byron J. Ma, Jun. “Learning to Rank for Hybrid Recommendation”. Natural Science Foundation of China
6. Li, Hang. “A Short Introduction to Learning to Rank” *Information-Based Induction Sciences and Machine Learning.* Microsoft Research, 10 October 2011
7. Cao, Zhe. Tao, Qin. Liu, Tie-Yan. Tsai, Ming-Feng. Li, Hang. “Learning to Rank: From Pairwise Approach to Listwise Approach” *A probabilistic Framework for Learning to Rank.* Proceedings of the 24th International Conference on Machine Learning, Corvallis, OR, 2007
8. Bappalige, Sachin. “An Introduction to Apache Hadoop for Big-Data”. Internet. opensource.com/life/14/8/intro-apache-hadoop-big-data

9. Zaharia, M., Chowdhury M., Franklin M., Shenker S., Stoica I. Spark: Cluster Computing with Working Sets, ACM, 2010
10. Shukla, Shilpa. Lease, Matthew. Tewari, Ambuj. “Parallelizing ListNet Training using Spark”. University of Texas at Austin. August, 2012, Portland, OR
11. Intel IT. Internet. “Using Apache Hadoop for Context Aware Recommender Systems” Intel IT White Paper. Big Data, February 2014
12. Blanco, Roi. Cambazoglu, Barla. Mika, Peter. Torzec, Nicolas. “Entity Recommendations in Web Search” Yahoo! Labs. 2013.
13. Yu, Xiao. Ma, Hao. Hsu, Bo-June. Han Jiawei. “On Building Entity Recommender Systems Using User Click Log and Freebase Knowledge”. Microsoft Research, University of Illinois at Urbana-Champaign. 2014.
14. Databricks Training. Internet. “Movie Recommendation with MLib” databricks-training.s3.amazonaws.com/movie-recommendation-with-mllib.html.