Master of Science Thesis

Investigating the impact of hybrid optimization strategies on distributed machine learning algorithms

Supervisor:
Prof. Dr. Markl, Volker

Author:
Gaur, Prateek

Advisors:
Heimel, Max
Boden, Christoph

August 10, 2014

Contents

Abstract

Many parallel data-driven systems have been successful in their ability to store and process large volumes of data. This has led to an increased interest in performing large-scale analytics on this data. Much acclaimed for its ability to scale petabytes of data, the MapReduce framework has been found to be limiting for iterative algorithms. Such iterative algorithms form the basis for many domains of data analysis. To address these challenges, various new techniques have been proposed. These usually revolve around either developing extensions to the existing systems or coming up with specialized domain specific systems.

Tackling this problem at an algorithmic level, we propose a set of optimization techniques that train either locally producing a sub-optimal, but a fast solution or globally creating slower yet optimal solutions. We evaluate the tradeoffs between these training approaches from the dimensions of quality and performance. Further, we suggest and investigate hybrid training techniques as a possible “middle ground" that try to come up with a better solution while still taking substantially less time than the global approaches.

Initial experiments have shown that the proposed architecture yields accurate predictions in a shorter training time following an easy-to-use framework. Our study aims to provide necessary guidelines to Data Scientists for choosing the most effective combination for the performance and cost requirements of a given learning task.

Keywords: stochastic gradient descent, online learning, ensembles, logistic regression, batch gradient descent, kmeans, global, local, hybrid, named-entity

Acknowledgements

I am indebted to my thesis advisors Max Heimel and Christoph Boden for their valuable ideas and continuous feedback. It started as a short discussion about their idea, and they ensured to help me see it come through to reality in the form of this thesis.

I would also like to thank my academic supervisor Prof. Dr. Volker Markl, the head of DIMA research group. I would like to recognize the efforts of Andre Hacker, Robert Metzger and Johannes Kirschnick who played a crucial role in not only solving the technical issues that I faced, but also ensuring that I understand the basics behind them.

Last but the not the least I am grateful to my friends and classmates Dipesh, Pino, Faisal, Janani, Suryamita, Silvia, Times and Hiroshi, who supported me till the end and made the entire journey a memorable experience. Finally, I would like to thank the whole IT4BI team and for giving me this opportunity to express myself and support me throughout this endeavor.

Foreword

This work is presented in partial fulfillment of the of the requirements for the degree of Master of Science in the subject of Information Technologies for Business Intelligence. This work was carried out at Database Systems and Information Management (DIMA) group at Technische Universität Berlin (TU Berlin), Germany.

Chapter 1

Introduction

The world now holds two times more bytes of data as there are liters of water in its oceans. It is also said that in the next five years, we will generate more data as humankind than we generated in the previous 5000 years [1]. This explosion of data is referred to as *data deluge* or *information flood*. Such a large volume of data, often dubbed as "Big Data", comes with its own set of of challenges. In the past few years, cost-efficient novel programming paradigms have emerged that make storing and analyzing this data more accessible. Out of the most famous one is MapReduce [2] from Google. The open-source implementation of MapReduce, Hadoop [3], has widely been adopted by various organizations. Its various advantages include the ability to horizontally scale petabytes of data on the available commodity servers, easy to understand & use programming semantics, good fault-tolerance and locality optimization.

With the advent of "big data", many problems that have previously been impossible are now a reality. For example, statistical machine translation only works well at large scale. Large-scale data analysis is a broad field that overalaps many disciplines such as machine learning, data mining and data science [4]. However, with the size of enterprise data reaching petabytes, data-analysis at large-scale has become a pressing problem [5]. This is mainly because the complexity of many traditional learning algorithms makes them difficult to parallelize. Initially, the move from batch to online learning algorithms was able to deal with the rapidly increasing dataset size, due to their linear run-time and fairly simple algorithmic complexity. But online algorithms weren't an ideal solution to deal with the dataset that extends more than a single disk. In 2006, Chu et al.

[6] proposed a general technique for parallel programming of a large class of machine learning algorithms. In their work, they showed that any algorithm fitting the *Statistical Query Model* [7] can be written in a "summation form" that can easily be expressed in MapReduce framework achieving linear speed-up with the number of cores. According to their approach, "throwing more cores" at the problem can substitute for coming up with new optimizations.

Problem. However, recently there has been a lot of research [6, 8, 9] to probe into the limitations of MapReduce and to explore the classes of algorithms that are not particularly suited to this programming model. The framework is found to be a poor fit for iterative algorithms, as due to its fixed execution pipeline and lack of general iteration support, each iteration has to be scheduled as a single MapReduce job with a high start-up cost. Also, due to its *shared-nothing architecture* [10], a lot of unnecessary network traffic is generated as all the static, iteration-invariant data has to be re-read from disk and reprocessed for each iteration, and the result of each iteration has to be materialized to the HDFS [11]. Algorithms that are not suited for MapReduce paradigm include iterative graph algorithms (e.g., PageRank), gradient descent (e.g., for training logistic regression classifier) and expectation maximization (e.g., KMeans).

This problem can be alleviated both at platform and algorithmic level. Various alternative platforms have been introduced that alleviate the iteration-based limitation of MapReduce by adding native support for iterations. Examples include: Microsoft's *Naiad* [12] and *Spark* [13]. There has been a study [14] that tries to solve this problem at algorithmic level. They claim, if Hadoop is a "hammer" then just throw away everything that's not a "nail". Here, "nails" refer to the problems that fit MapReduce paradigm. So, if we find "screws", then instead of inventing a screwdriver, we just get rid of the screws. Specifically, to target non-fitting iterative algorithms in MapReduce, a simple solution could be to avoid iterations instead of inventing a new framework and incur the cost associated with switching tools and various other engineering costs. The argument is further cemented by the successful adoption of the above technique at Twitter [15].

In this work, we evaluate different approaches to scaling out both KMeans and Logistic Regression in a significant big-data setting. These procedures are evaluated from 2 different points of views: 1) parallel algorithm and 2) the distributed platform. From the

point of view of platforms, we compare two well known distributed platforms: ***Apache Hadoop*** and ***Apache Flink*** [16]. Hadoop uses MapReduce programming model and HDFS for storage. Apache Flink provides native support for iterations and supports HDFS as well. It uses PACT [17] programming model. From an algorithmic point of view, we compare the distributed versions of their traditional implementations against their suboptimal approximations in order to avoid the switching costs as proposed in [14]. Also, the actual training can either be carried out locally or globally. *Global* training methods exploit the entire dataset, giving the exact solution but are fairly resource intensive and slow as they suffer from the inescapable cost of storing and communicating the static, iteration invariant data. While, in *local* training methods, the real training happens within local partitions, which may sacrifice accuracy but helps in eliminating the inherent communication overhead making them faster. In this study, we also explore *hybrid* training techniques that use the approximate solution from a local training as a starting point for global training thereby providing a reasonable speed-up from local methods along with maintaining the accuracy of global methods. This study doesn't focus on comparing Flink with Hadoop as a lot of existing studies have already shown that Flink performs better than a lot of popular execution engines including Hadoop [16]. Comparisons with Flink are merely to ensure that our hypothesis works independent of any platform choice.

Goal. This study reviews and examines the inherent trade-offs in terms of quality and performance in *local* vs.. *global* training strategies. Furthermore, we suggest and investigate the *hybrid* optimization as a "middle ground". The evaluations are carried out from the point of view of both the *database experts* whose main focus is to get the results as fast as possible and *statisticians* who want to get the most accurate results within a given error frame. The study aims to provide the necessary insights to both the enterprise-owners and data-scientists/ designers for selecting a solution that best meets the cost and performance requirements for the learning task at hand.

Road map. This thesis is divided into 6 chapters. Following this introduction is Chapter 2, which presents the background behind all the concepts that we use. In particular, Section 2.1 introduces the platforms that we use, Section 2.2 talks about the entire theory behind large-scale predictive analytics and associated algorithms and

Section 2.3 presents the summary of the related work that we've followed. After this, Chapter 3 introduces our approach in detail. This is followed by Chapter 4 which discusses how we designed and implemented the various algorithms. The experimental phase is summarized in Chapter 5. There we talk about our experimental setup, datasets, scoring methods and the type of experiments performed. Chapter 6 is a summary of the thesis that describes our contributions including the conclusions we have drawn and the ideas related to our future work. Following the concluding chapter is our Appendix which contains some miscellaneous information which doesn't fit elsewhere. Concluding this thesis is the bibliography.

Chapter 2

Background and Related Work

This study lies at the intersection of Large Scale Machine Learning and Parallel Data Processing. We discuss how various problems faced in predictive analytics can be resolved using some fundamental heuristics from the field of distributed computing. In this chapter, we introduce these concepts independent of each other. We start with discussing about the fundamentals of Parallel Data processing by introducing Apache Hadoop and Apache Flink as our chosen computing platforms. Further, we introduce the basics of machine learning and the key concepts behind the algorithms that are implemented. As the area in itself is vast, we only cover some topics in detail, which are needed for the purpose of our study. A short survey of related work is given at the end of this chapter covering some of the key research done in our domain.

2.1 Computing Platforms

Both the distributed platforms in consideration have their set of pros and cons when it comes to the implementation of machine learning algorithms. **Apache Hadoop** [3] allows for distributed processing of large dataset across clusters of computers using simple programming models. The computation paradigm behind Hadoop is called *MapReduce* [18], which is easily parallelized. Hadoop uses *HDFS*, its own distributed file system, which is designed to handle node failures in an automated way (just like MapReduce). This allows Hadoop to support large clusters using commodity hardware. **Apache Flink** [16] is a cluster computing platform with powerful programming ab-

stractions, a high-performance runtime, and automatic program optimization and also provides native support for iterations. Table 2.1 summarizes the basic features of these platforms

	Hadoop	**Apache Flink**
Computation Paradigm	MapReduce	PACT
File System Supported	HDFS, FTP, Amazon S3 [19]	Local Files, HDFS, Amazon S3
Design Concept	Key-value Pairs	Record and Graph Oriented
Fault Tolerance Technique	Job Restarts, Redundancy	Job Restarts
Programming API(s)	Java	Java, Scala
Evaluation Technique	Speculative	Lazy

Table 2.1: Platform Comparison: Hadoop vs.. Apache Flink

Apache Mahout [20] runs on Hadoop and is a scalable machine learning library. It is a collection of many important algorithms for clustering, classification and collaborative filtering. Being highly optimized, it achieves a higher accuracy for non-distributed algorithms as well.

2.1.1 Hadoop

The most famous and widely used platform popularized by Google's J. Dean and S. Ghemawat is *MapReduce*. The best explanation on MapReduce is available in the original paper [2] and unfamiliar readers are advised to refer to it, for further details. Moreover, a basic understanding of the MapReduce paradigm is assumed in the rest of the work.

Hadoop stack is quite rich in terms of the support for complex data processing. *Streaming MapReduce* [21] provides the functionality of "pipes" in order to ease the development in the programming language of your choice but provides slightly lower performance and less flexibility than the native Java MapReduce. *Pig* [22], a high-level language developed by Yahoo provides functionality to handle batch data flow workloads. It is used for expressing data analysis and infrastructure processes. *Hive* [23], a

sql interpreter from Facebook includes a meta-store functionality to map files to their schemas. It adds a data warehouse functionality to the hadoop cluster. *Oozie* [24], is a PDL XML-based workflow engine that helps in creating a workflow of jobs.

HDFS. The **H**adoop **D**istributed **F**ilesystem (HDFS) is a scalable, distributed file system inspired by Google File System [18]. It stores the files in a shared-nothing cluster using the commodity hardware. When a file is copied to HDFS, it is divided into blocks, whose size can vary and is typically either 64 MB or 128 MB and then these blocks are stored on a few nodes depending on the block replication factor. The replication factor can be chosen beforehand and its default value is 3. A higher value increases the fault tolerance but comes with a space trade-off and also requires more loading time. Figure 2.1 shows an example of how data is stored in HDFS with a default replication factor. Apart from distributing the data across the data nodes and managing replication for redundancy, it is also responsible for the administrative tasks like adding, removing and recovery of failed data nodes. To summarize, HDFS serves as an API that improves data locality for higher level systems by allowing them to see exactly where the blocks are stored which eases the move of computation to the data.

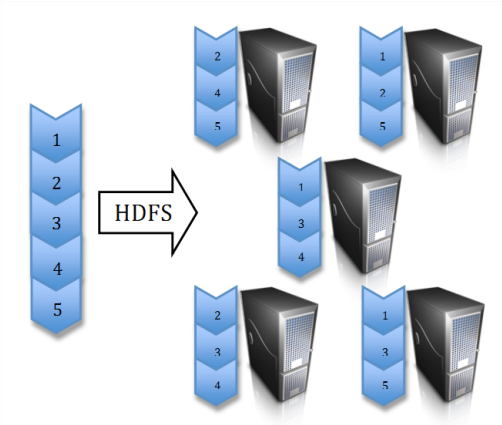


Figure 2.1: HDFS: Storage Example, picture adopted from [25]

MapReduce. The programming model consists of two basic second-order functions called ***Map*** and ***Reduce*** with an optional ***Combine***. Writing a MapReduce program involves writing the first order functions (custom implementations) for the second order Map and Reduce operators. These custom implementations are often referred to as

User **D**efined **F**unctions. Figure 2.2 summarizes the architectural flow of a MapReduce program. To illustrate the semantics of the programming model, we show the example of counting words in a document. The Map UDF is called for every input split and outputs the result in the form of key-value pairs. For example, the UDF receives a line containing a text that is then tokenized, and each single word is extracted. The Mapper then emits the word as the key and "1" as the value. The system then performs a *groupby* operation on the keys and sends the output to the appropriate reducer. This is usually done by applying a hash function on the key whose resultant value determines which reducer is chosen. Once the last mapper is finished, the reduce function is called for every key, which are unique words for our example, and an iterator over all the values of that key is also passed to this reducer. All this happens in the **shuffle phase** of the process. The *groupby* operation before the reduce phase is achieved by sorting the keys on every node that ensures that the reducers are called according to the sorted keys. For our example, the reduce UDF sums up all the values ("1") for a particular key and thereby, outputs the count for the respective words. The output of the reduce function is the final output that is then stored on the HDFS. Additionally, in the shuffle phase, a Combiner UDF could have been used to do the partial aggregation of the results from every mapper that can help in reducing the number of key-value pairs that are transferred over the network.

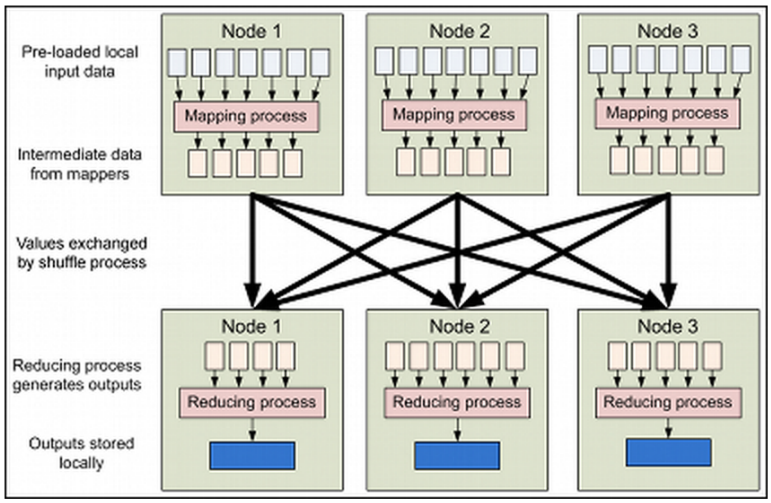


Figure 2.2: MapReduce Framework: Architecture, picture taken from [26]

In general, any problem mapped in MapReduce framework follows a similar ***static dataflow***, i.e. Map, combine (optional), reduce and materialize to HDFS. Most often such a simple flow is limiting for many real-world applications. In such cases, multiple MapReduce jobs are chained together to solve a single problem. Additional constructs such as Distributed Cache are also supported, but they do not affect the dataflow in any way. A ***Distributed Cache*** [27] is designed to broadcast small files to local disks of all the nodes to make them available across all the UDF calls. Its efficiency stems from the fact that these files are copied only once per job and are available for all the tasks on the node. For further details on distributed cache, please refer [3].

Due to task distribution across many nodes, hadoop suffers from the problem that a few slow nodes can limit the rate for the entire program. This is referred to as **Straggler effect**, which happens when a machine takes unusually long time to complete the last few map or reduce tasks. This could be because of some software-related misconfiguration, hardware degradation or poor clustering scheduling system. The main issue here is that the problem itself is hard to detect as the task still finish successfully but take longer than expected.

For such cases, instead of diagnosing and fixing the slow running tasks, hadoop tries to identify when a certain task is running slower than expected. Once the task is detected hadoop launches a backup task. This is referred to ask **Speculative Execution.** Whichever task finishes first becomes the definitive copy, and other simultaneous tasks are stopped.

2.1.2 Apache Flink

This section briefly covers the principal components of Apache Flink platform [28]. For a detailed description, readers can refer [16, 29, 30]. Apache Flink is a massively parallel data processing system. The essence of PACT programming model lies in its support for *flexible data flows*. In contrast to MapReduce, Apache Flink's operators such as Map, Reduce, Union and so on can be chained together in a directed acyclic graph (DAG) so that even more complex algorithms can be expressed in a single data flow. The major focus of this report is to check the correctness of our Hypothesis on hadoop and

on a platform that improves the major drawbacks of Hadoop. In this section, we will discuss its architecture, programming model and some other properties that make it an attractive computing platform for running iterative machine learning algorithms. The 2 major components of Flink's software stack are its programming model and parallel execution engine called *PACT* [17] and *Nephele* [31], respectively.

PACT programming model is based on *Parallelization Contracts* often abbreviated as PACTs. PACTs are second order functions just like Map and Reduce supported by Hadoop.[1] As of stable release 0.4; Flink supports Map, Reduce, Join, Cross, Union and CoGroup. Like in MapReduce, users have to write customized first order functions called UDFs. PACT uses *record data model* that allows every operator to take zero or more records as input or output. A Record can be compared to a database row containing objects from supported data types.[2] The fields of PACT records can be accessed by their indexes only due to the lack of inbuilt schema. We discuss the basic ones below:

Map and Reduce are similar to Hadoop. However, as mentioned earlier, due to missing schema, the only way to access the fields is through indexes. So in Reduce, the user has to explicitly specify the index of the chosen record which holds the key to be used to groupby the record with. All the other operators that use key should follow suit.

Cross is used to perform the cross product for two inputs and every possible pair of records from different inputs is the input for the UDF.

Join is used to perform Inner Join for two inputs as known from relational databases. A UDF gets called for only those pair of inputs that have the same key.

CoGroup can be compared to Reduce but has multiple input points instead of a single one. A UDF gets called for every group of records with the same key. Here, the UDF receives two iterators with records from the first and the second input.

[1] In the newest stable release, operators are referred to as Transformations

[2] The newest release of Flink uses the concept of datatypes over records.

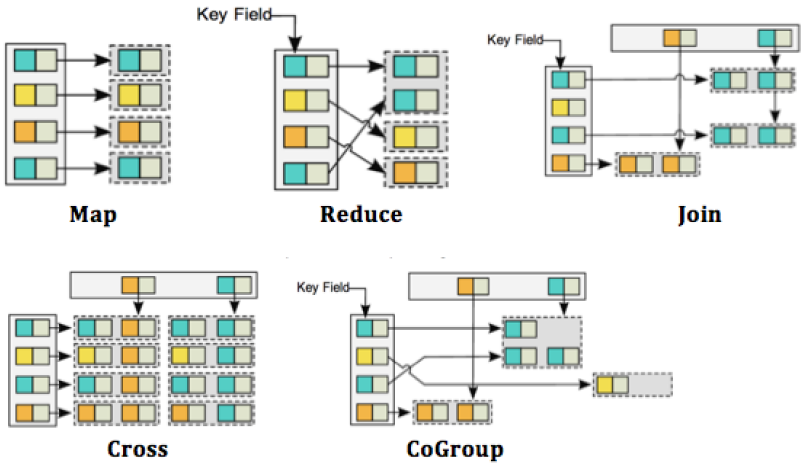


Figure 2.3: Flink Operator Semantics adopted from [32]

Flink follows *lazy evaluation strategy* whereupon running the program's main method, the data loading and transformations do not happen directly. Instead, each operation is created and added to the program's plan. The operations are executed upon invocation of execute() method where the program is transferred to the PACT compiler.

Once the plan is passed to the PACT compiler, it transforms the plan into a Nephele *job graph*. Then, Nephele parallel execution engine executes this job graph. A Nephele Job Graph (also a DAG) is an optimized version of the PACT plan. This optimization is achieved by optimizer, which is the central component of the PACT compiler. Just like a database optimizer, the main task of the optimizer is to find the execution plan that minimizes the overall costs of network communication and disk I/O. Figure 2.4 shows how the entire process is carried out for a Join, i.e. Join operator.

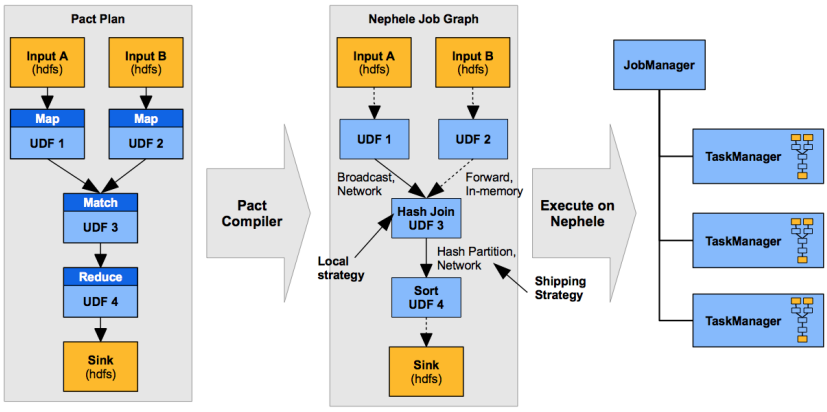


Figure 2.4: Execution of Pact Plan, adopted from [31, 27]

Finally, Nephele spans the optimized job graph over the nodes in the cluster and executes the job. The execution is carried out by a single master process called *JobTracker*, which manages resources, handles failures and schedule jobs and many *TaskTracker* processes where the actual execution takes place.

In Flink, iterative programs are executed by defining a **step function** that consumes the input and computes the next version of the intermediate solution. This function is then embedded in a special operator. The operator repeatedly invokes the step function on the current state of the iteration until a certain terminal condition is reached. Hence, the entire strategy of looping is kept inside the system's runtime, rather than the client's. Loop-invariant data is automatically cached into the memory [33]. Another major point to note here is that Flink currently doesn't support any *fault tolerance* apart from restarting the failed jobs. The support for Distributed Cache has also been added in the subsequent releases.[3]

[3]In the latest release, the concept of broadcast variables is introduced that are more efficient and easier to use, please refer to the Javadocs for further details

2.2 Machine Learning

2.2.1 Basic Concepts

The classification problem is one of the most-known problem in the field of data mining [34]. In this report, the focus is on the use case of *named-entity recognition* to introduce the basics of Machine Learning. *Named Entities* (NE) are the phrases containing the names of persons, locations, and so forth. They are important for the access to the document content as they form the building blocks upon which the entire document analysis is based [35].

In a textual document, an NE can be found accompanied by its **contexts**, i.e. the words that are to the left or the right of it. These contexts can be used to understand the entity represented by the word. For example, if the word "President" occurs in a text then it is highly likely that this word or context may be followed by the name of the president like President "Obama." The same is true for a word that is preceded by the string "footballer," it is highly likely that it will be followed by the name of a footballer [35].

Classification. In classification, a *labeled* training dataset is used in order to correctly classify the unlabeled data samples based on their unique properties called *features*. Once, we have the input converted to numeric features, we can use any status-quo machine learning algorithm to learn a model that allows us to predict the NE class belonging to a particular word. This is called *Supervised learning* technique due to the existence of different labels in terms of the NE Class. In our example for the use-case of Named-Entity Recognition, our labels are the possible named-entities such as persons and books. The feature set in our case include the word along with its surrounding words such as for *President Obama*, both President and Obama are the features used to predict whether the phrase President Obama refers to a person entity or not. It is a *Multi-class* classification problem as there could be a lot of discrete labels/ entities to which the word can belong. We can reduce this problem to a *Binary classification* problem by choosing whether the word belongs to a particular entity or not. Here, the target variable is binary as it only contains zero or one.

In general, the model we aim to learn is a function $h : X \to Y$, called *hypothesis*,

that maps the word to the category 0 or 1 based on its existence as the chosen entity. Theoretically, the learned hypothesis h should be as close as possible to the target function $t : X \rightarrow Y$ that always makes the correct prediction but is impossible to learn and is chronically unknown for most of the real world problems [27]. The *error function* or the *loss function* helps in computing how close we are to the target function by using the prediction and the true labels. *Residual Sum of Squares* is the most common and intuitive error function [27, 36].

$$RSS = \sum_{i=1}^{N} (y_i - h(x_i))^2 \tag{2.1}$$

In many cases, different NEs can be used using the same context. For example, the context "Mr." can precede both Zidane or Obama making it difficult to decide whether it refers to a president's name or a footballer's name. In such a case, a common strategy is to introduce the concept of Majority Voting by adding a vote field with each context$_i$ in the document. The value of the vote is incremented with the weight of the context, each time the context$_i$ is encountered [37]:

$$vote_i = vote_i + w_i \tag{2.2}$$

The algorithm chooses the final entity by comparing the value of vote$_i$ with the votes received by the rest of the contexts [35].

Clustering. The process of clustering is similar to that followed in classification. However, in clustering, the training dataset is not labeled. This is true for many real world datasets. This is because annotating a dataset is usually an arduous task. In such cases, we use the contexts of the words to check for their similarity with an assumption that words that are similar belong to the same group as compared to the words that are not similar. These groups are often referred to as clusters. We can select the initial number of clusters based on our assumption about the number of groups that exist in the dataset. Such an approach is often referred to as an *UnSupervised Learning approach.*

Feature Extraction. The process of transforming raw input data to a reduced representation set of features (also named features vector) is called *Feature Extraction.* This is usually done to avoid the redundancy when the dataset is too large to be processed.

The feature-set is chosen so as to represent the entire dataset, thereby, reducing the need to use the entire dataset for the task at hand.

Feature Selection. Sometimes a single feature alone is not a good predictor, for example, neither height nor weight is good enough to prove Obesity. In such cases, we look for combinations of features that are discriminatory. For example, Height and weight taken together predict Obesity fairly well. Hence, by selecting the appropriate set of features, we can do a good job of classification. This is regarded as the process of *Feature Selection*. Both of the above-mentioned features are linear in nature (their increase or decrease has a similar effect on the class, hence they combine linearly). These linear combinations can easily be plotted on a line and can be solved using the equations (for example, Principal Component Analysis [38]) but there are some combinations that are non-linear in nature. Let's take an example of distinguishing aircrafts from cars using their weights as the predictor. If the vehicle is hefty then it is mostly an aircraft. However, there are certain small planes that weigh less than a car. So, if the vehicle weighs between one to two thousand pounds, it is probably a car after which it falls into the category of a light jet. In such cases, the only way to find discriminatory combinations is to search exhaustively through the entire combination space. The number of combinations increase with dimensionality and so does the space and time complexity in looking for discriminatory combinations.

Generalization Error. A good generalization behavior is the one where the hypothesis in consideration correctly approximates the target function for unseen data. To achieve this, a model, with enough complexity, could simply memorize the complete training dataset. Such a model often under-performs for unseen documents. This is called the problem of *Overfitting*, where the model fits the training data too much which often leads to memorizing all the irregularities and noise as well. Here, noise can also refer to the irrelevant features that might improve the accuracy for the training data but only have a little statistical significance. *Underfitting* occurs when the model is not complex enough to capture the underlying trends. Intuitively, this problem can be targeted by increasing the complexity of the model, i.e. by adding more dimensions and so forth.

The generalization behavior of a training model can only be known after it is applied to the unseen data. To get an earlier approximation, the input data can be split into two

parts- for *training* and *testing* and the analysis can be performed on the former while the validation of the analysis can be performed on the latter. This is called the *Cross Validation* process. To improve the robustness of results, since the size of training data is too small to get useful insights, multiple rounds of cross-validation are performed using different partitions and the results are finally averaged over all the rounds. The rest of the chapter covers two popular and simplistic supervised and unsupervised learning algorithms- ***KMeans*** and ***Logistic Regression***. The reason behind their popularity lies in their effectiveness despite a simpler underlying concept.

2.2.2 KMeans

KMeans algorithm (or Lloyd's method) is one of the most-popular clustering algorithm known since 1950s. It works by choosing k points out of n as initial centers. Then the algorithm performs the following steps, iteratively:

1. each point is assigned to the cluster whose centroid is closest to it
2. existing centroids are recomputed by computing the geometric average of the existing points in the cluster

Although, numerous variations and extensions of KMeans are available [20] such as K-Medoids, Fuzzy KMeans, Spherical KMeans, using kd-trees to speed up each KMeans step and so on. However, in this report, we are particularly concerned with two relatively new extensions: Ball KMeans (variant) and KMeans++ (initialization). Algorithm 1 summarizes the basic KMeans algorithm.

Algorithm 1: Basic KMeans Algorithm [4]

```
1  Centroids ← select k points ∈ Points
2  while not done do
3      for c ∈ Centroids do
4          Clusters[c] ← 0
5      for p ∈ Points do
6          d_min ← ∞
7          for c ∈ Centroids do
8              d ← dist(p, c)
9              if d < d_min then
10                 d_min ← d
11                 c_min ← c
12         Clusters[c_min] ← Clusters[c_min] ∪ {p}
13     Centroids ← 0
14     for C ∈ Clusters do
15         Centroids ← Centroids ∪ {mean(C)}
```

The output of the algorithm gets affected by: 1) centroid initialization, i.e. how and how many centers are selected initially, and 2) stopping condition, i.e. how long should we continue with our iterations. Let us go through some possible variations that can help in positively affecting the final output by individually taking care of the above constraints.

2.2.2.1 Initialization

Considered as the single-most important factor in getting a high-quality clustering, *initialization* or *seeding* refers to the process of selecting the first k points to become centroids. Two basic initialization strategies are discussed below:

Random Selection

In this technique, k points are selected uniformly at random. It produces a seed but has no real guarantees in terms of representing the actual points. Therefore, such a selection leads to different clustering outcomes. Assuming there are k real clusters, an ideal initialization often refers to a situation where the seeds come from those k clusters and no two seeds come from the same cluster. In such a situation, the clustering algorithm will set nearly all the points to their right clusters in just a few steps as we have initially

assumed that the clusters do exist in the data.

Random Selection based initialization fails in a situation when two seeds are selected from the same real cluster, causing it to split between two KMeans clusters. Like any machine learning technique stuck at its local optima, such a situation can be tackled by leveraging ***multiple-restarts*** that make it impossible not to have a good set of initial centroids [4].

KMeans++

In general, the initialization of centroids is done randomly. However, such random initialization often leads to different clusters making it difficult to benchmark the results. Intuitively, selecting the seeds that are as far apart to each other as possible should eliminate the problem of having two seeds in the same cluster thereby improving the quality of initialization.

Algorithm 2: KMeans++ Initialization Algorithm [4]

Input: set C of weighted centroids from first pass, target number of final clusters k

Output: set S of initial centroids of final clusters

```
S = 0
foreach c_i ∈ C do
    d_i = ∞
while |S| < |C| do
    sample s_new ~ D
    S = S ∪ s_new
    foreach c_i ∈ C do
        d_i = min(d_i, |s_new − c_i|)
return S
```

Algorithm 2 is based on [39]. It works by selecting the initial center uniformly at random from all the data points. Then the distance between this center and all the points is computed. The computed distances are then arranged using a weighted probability distribution. A new center is then chosen randomly with a probability proportional to the square of its distance from the initial center. All the steps are repeated until k centers are selected. These are then passed as input the to KMeans algorithm.

2.2.2.2 Number of Clusters

While discussing the problem above, we assumed that we are given n d-dimensional points and the distance measure $dist$, we also assumed that we are given the initial k.

This assumption holds true in theory but in practice, when we try to cluster we cannot know what the value of k actually is. That is the reason it is called an *unsupervised learning* problem, because there is no ground truth.

However, there exists one well known way of addressing this, the **Elbow Method**. In Elbow method, one tries to cluster the given dataset with different values of k and plots the total cost T_c as a function of k. In general, the total cost goes down as the number of clusters increase because they tend to reach the number of "real" clusters in the dataset. Intuitively, this trend should follow, as the data representation becomes more logical, until the cost plateaus which is when the clusters fit the real clusters exactly [4].

2.2.2.3 Distance Measure

Normally, the choice of distance measure is not fixed and varies from one designer to the other. In general, most papers on clustering work with the status-quo *squared Euclidean distance* (also squared L2-norm). For robustness in our implementation, we support various other distance measures such as Manhattan distance and Cosine distance. However, in practice, most distance measures used are variants of the squared Euclidean or cosine distance, which lead to similar results.

2.2.2.4 Quality

When it comes to Unsupervised Learning, there is no correct way to measure the correctness of the underlying algorithm. It is often said- "*clustering is in the eye of the beholder*" [4, 40]. As, there is no "right answer", in such cases, a better way to go is to measure the quality of the results. Coming up with a metric for measuring quality is an arduous task. Intuitively, we strive for:

- *compactness*, i.e. small intra-cluster distance between any two points (points lying in the same cluster should be close to each other)
- *disjointness*, i.e. large inter-cluster distance between any two cluster centers (two

different clusters should be relatively far away from each other)

So, the status-quo "total-cost" is a good starting point. The *Dunn-Index* and *Davies-Bouldin* Index try to express compactness and disjointness in one score. These are called "*Internal Scores*" because they often measure the intra-cluster properties. We will discuss quality measures in further detail in section 5.3.

2.2.2.5 Convergence

The main KMeans step is performed multiple times before some convergence criteria is met. The algorithm is stopped after:

- no change is observed in cluster assignment
- a fixed number of iterations
- a quality metric plateau, like total cost, i.e. the quality metric stops decreasing after a few iterations

2.2.2.6 Ball KMeans

A clustering technique [41] proved better for highly clusterable dataset. It works by trying to find initial centroids in each "core" of the real clusters. It also tries to avoid outliers in centroid computation. Algorithm 4 discusses the basic algorithm behind Ball KMeans clustering.

Algorithm 4: Ball KMeans Algorithm [41]

Input: set C of weighted centroids from first pass, set S of k centroids
Output: set S of adjusted centroids of final clusters

foreach $c_i \in C$ **do**
  $n_i = argmin_j \mid c_i - s_j \mid$
foreach $s_j \in S$ **do**
  $t_j = \{c_i \mid n_i = j\}$
  $x_j = median \mid t_j - s_j \mid$
  $s_j = mean\{x_j \mid x_j > \mid t_j - s_j \mid\}$
while $\mid S \mid < \mid C \mid$ **do**
  sample $s_{new} \sim D$
  $S = S \bigcup s_{new}$
  foreach $c_i \in C$ **do**
    $d_i = min\,(d_i, \mid s_{new} - c_i \mid)$
return S

2.2.2.7 Nearest Cluster Search

The most-important step in any KMeans implementation is about finding the nearest cluster. As discussed above, in Local-KMeans implementation this problem is further magnified as the search has to happen over a lot more than k clusters. One way [41] to decrease these searching costs is to use *random projections* [42] to reduce the problem of finding a nearby cluster to a one-dimensional search. As there is no guarantee of finding the nearest cluster even upon examination of several candidates, the results from several projection searches can be combined. In this study, we use the *Brute Force technique* where search is performed using all the dimensions. To go further in detail about various techniques for Approximate Nearest Neighbor search, please refer [43].

Algorithm 5: Projection Search: Algorithm [41]

$Candidates \leftarrow 0$
for $p \in Projections$ **do**
  $pq \leftarrow$ project query on to p
  $c \leftarrow$ closest neighbor of pq induced by p
  $Candidates \leftarrow Candidates \bigcup$ ball of size b around c
keep closest k from $Candidates$ by computing the actual distances

2.2.3 Logistic Regression

Thanks to its linear model, Logistic Regression has widely been used in the application of machine learning. It has also been adopted in other fields like economics, social science or medicine [44].

Unlike other complex classification algorithms, logistic regression has a simple underlying linear model. A linear model is a linear combination of x of the form:

$$h_w(x) = w_0 + w_1 x_1 + ... + w_D x_D \tag{2.3}$$

where $h_w(x)$ is the hypothesis and can be used directly for prediction in *linear regression* [45]. w_0 is called *bias* or intercept term and defines a fixed offset in the prediction. If we assume $x_0 = 1$ then the dot product $w^T x$ can be used to express the linear hypothesis without having to exclude the bias parameter.

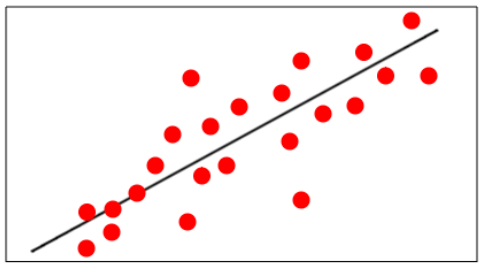


Figure 2.5: Linear Regression Classifier: Continuous Attributes

Linear regression learner works well for cases where both the predictor and the target variables are continuous. Example, let us assume we want to predict how increase in the amount spent on advertisements affects the sales of the product. As you can see from figure 2.5, a linear regression can fit well for such a case. But, what happens when the target attribute is not continuous? Suppose, we do a qualitative study to check whether the increased sale is due to the ad campaign and the response is either Yes or No (encoded as 0 or 1). In such cases, there is no gradual transition, i.e. the target variable jumps from one outcome to another. As seen in the first diagram of figure 2.6, a straight line model is a poor fit for such a usecase.[4]

[4]The example is adapted from a blog post entry at [46]

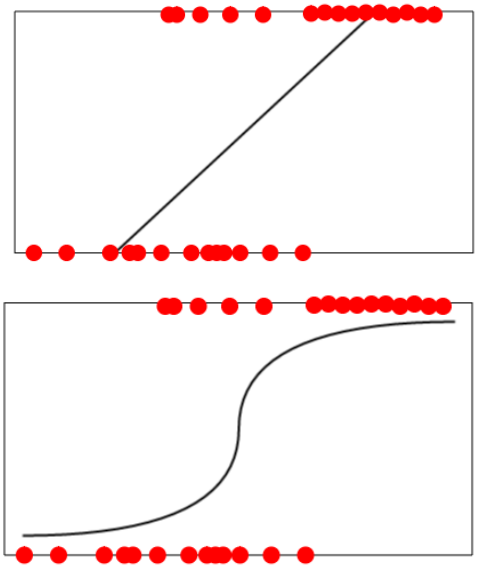


Figure 2.6: Linear Regression and Logistic Regression models applied to binary classification problem with discrete target variables.

However, the *S-shaped* curve in the second diagram of figure 2.6 better fits the data. This approach is called ***Logistic Regression***. Logistic Regression works on the mathematical principal that transforming the target variable y_i to the **logarithm of the odds** of y_i will introduce linearity between x_i and y_i. For Logistic Regression, the linear model $w^T x$ can be fed into the (s-shaped) logistic function σ which is also known as a *sigmoid function* [47]. So, the likelihood that the sample x belongs to the positive class 1 can be expressed as:

$$h_w(x) = P(y = 1) = \sigma(w^T x) = \frac{1}{1 + e^{-w^T x}} \tag{2.4}$$

The size of the weight vector w is initialized to the length i of the feature vector. The coefficient of the w_i represents the impact of feature i. Hence, having $w_i = 0$ is typical for irrelevant features [27] .

In this paper, we are trying to solve a binary classification task. Given a training dataset $X = \{(x_i, y_i) : i = 1.....,n\}$, where $x_i \in R^d$ are the input data points while $y_i \in \{-1, 1\}$ are the corresponding labels. The task of the classifier is to learn the model w that best fits the data. This can be achieved by minimizing the error function, which

in general terms means trying to fit the training data in the best-possible way. One of an important task of the designer is to choose a proper error function. For our experiments, we use *maximum likelihood estimation* which tries to find a model w that maximizes the likelihood $L(w)$ [48]. It is equivalent to finding a model w that minimizes the negative log likelihood $-l(w) = -ln(L(w))$. Solving the derivative, it leads to an optimization problem which can be expressed as [27]:

$$arg\ max_w - l(w) = arg\ max_w - \sum_{i=1}^{N} y_i ln(h_w(x_i)) + (1 - y_i) ln(1 - h_w(x)) \tag{2.5}$$

This problem doesn't have any closed form solution, hence, other training techniques are applied over logistic model to help in solving the optimization problem. These techniques range from iterative techniques of Gradient Descent and Newton-Raphson [49] to the online techniques of Stochastic Gradient Descent and LBFGS [50]. These techniques usually formulate the problem as a convex optimization problem which has a closed form solution. More on convex optimization in chapter 3. In the next part, we will go into the detail of Gradient Descent (or batch gradient descent, as it is generally referred as) and Stochastic Gradient descent.

2.2.3.1 Batch Gradient Descent

As shown in equation (2.5), solving a machine learning problem with Logistic Regression can easily be cast as a functional optimization problem that is trying to reduce the associated empirical risk. Summarizing the discussion in section 2.2.3, empirical risk is basically trying to minimize the functional prediction $h(x_i)$ by comparing it with the actual output y_i which looks like:

$$\frac{1}{n} \sum_{i=0}^{n} l(h(x_i), y_i) \tag{2.6}$$

To ease the derivation, lets assume the family of functions in hypothesis space are parameterized by vector θ. So, the minimization equation looks like:

$$arg\ min_\theta\ \frac{1}{n} \sum_{i=0}^{n} l(h(x_i, \theta), y_i) \tag{2.7}$$

This optimization problem can often be solved using *Batch Gradient Descent* approach. To simplify our notation, we can rewrite equation 2.5 as $arg\ min_w L(\theta)$. Now, the gradient of L can be written as:

$$\nabla L(\theta) = \left[\frac{\partial L(\theta)}{\partial w_0}, \frac{\partial L(\theta)}{\partial w_1}, ... \frac{\partial L(\theta)}{\partial w_d}\right] \tag{2.8}$$

where $\nabla L(\theta)$ refers to the gradient of L. It represents a vector field whose direction points towards that of increasing L and the magnitude refers to the rate of that increase. A single step in the direction opposite to the gradient from a point $a \rightarrow b$ can be represented as $b = a - \gamma \nabla L(a)$, then $L(a) \geq L(b)$ as long as the learning rate $\gamma > 0$ [14]. Repeating the above process, a gradient update rule is defined as:

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \gamma^{(t)} \nabla L(\theta^{(t)}) \tag{2.9}$$

So, in the end, we can be sure that the sequence converges to the desired local minimum.

$$L(\theta^{(0)}) \geq L(\theta^{(1)}) \geq L(\theta^{(2)}) ... \tag{2.10}$$

If the loss function is convex and γ is chosen carefully, we are guaranteed to converge to a global minimum. More on Convex optimization in chapter 3. In figure 2.7, we can see a contour plot for gradient descent showing the iterations to a global minimum.

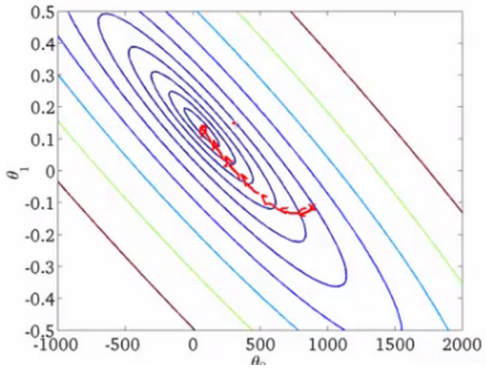


Figure 2.7: Batch Gradient Descent Training: Contour Plot

Checking for convergence in Batch gradient descent is fairly intuitive. We can plot

cost function against the number of iterations. As we know that the cost should decrease on every iteration, we can check whether the cost plateaus when compared against the previous iteration.

2.2.3.2 Stochastic Gradient Descent

Cost function optimization using Batch Gradient Descent is very precise, but the summation step makes it limiting for larger datasets. For such cases, training using Stochastic Gradient descent is preferred. Originally, under the name ADALINE [51], stochastic gradient descent is a popular training technique for many existing learning models such as Linear/ Logistic Regression, Support Vector Machines [52], artificial neural networks [53] and so forth.

Stochastic gradient descent is the online variant of the classical gradient descent training methods. Subgradient techniques are those in which instead of using all the training examples only a subset of those are used for gradient computation that leads to approximation in results. In stochastic gradient techniques, this subset is reduced to a single training instance. Algorithm consists of 2 basic steps:

1. Randomly shuffle or reorder the training samples
2. Perform the actual training on every sample.

Algorithm 6: SGD ($\{c^1, ..., c^m\}$,T,η, ω_0) [5]

for $t = 1 \rightarrow T$ **do**
    Draw $j \in \{1....m\}$ uniformly at random
    $w_t \leftarrow w_{t-1} - \eta d_w c^j(w_{t-1})$
return w_T

Figure 6 shows the pseudo code for the algorithm. We start by looking at the first example and take a step according to the cost of just the 1st training example, after which we move to the second one. Now, the algorithm tries to fit the second training example in the parameter space, without looking at any other training example. The process continues until it reaches the end of the data. The entire procedure can repeat by taking multiple passes over the entire dataset.

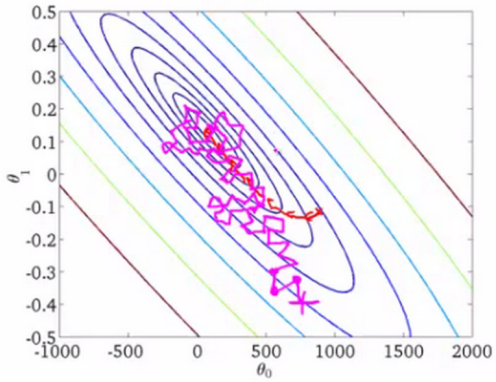


Figure 2.8: Stochastic Gradient Descent Training: Contour Plot

As can be seen in figure 2.8, with stochastic gradient descent based training, unlike batch gradient descent, every iteration does not always move in the direction of global minimum. Although each iteration is faster but it is only seeing one training example at a time and tries to do what can be the best for that instance. In practice, we may need to loop over the entire dataset 1-10 times for smaller datasets. However, if the dataset is truly massive, it is possible that a single pass is sufficient for a perfectly good hypothesis.

The initial step of *Random shuffling* is considered as a good practice. It is done to avoid the possible bias introduced due to ordering of the data. It has been observed [5] that randomization speeds up convergence by a little bit. Stochastic Gradient Descent never actually converges like batch gradient descent but more often than not it ends up wandering in the good neighborhood of the global minimum. But, unlike Batch Gradient Descent, we cannot actually plot the averaged cost function against the number of iterations. However, we can plot the cost function (squared error) for every single sample and can plot it before updating the model. In general, it can be done after every fixed number of samples. This helps in deciding the ideal learning rate to chose because sometimes we need to tweak the learning rate to reach closer to the global solution.

2.2.3.3 Ensemble Methods

In Ensemble based learning, multiple learning algorithms are used to obtain better predictions than can be obtained from the single underlying model itself. The learning process follows 2 basic steps:

1. Learn multiple alternative models for a single concept using separate training datasets and/ or different learning algorithms.

2. Combine the decisions from these multiple models, e.g., by using weighted voting.

The fundamental idea behind the value of Ensemble based learning is quite general. Let's say we want to count the number of coins in a jar. Although, an individual's vote will be mostly right but an averaged decision of the group (human ensembles) will be considered more reliable. Similar is the case with the classifiers. When independent and diverse decisions of multiple learners each of which is more accurate than random guessing are combined, they cancel out each other's random errors, and correct decisions are reinforced [37].

There are two types of Ensembles: *Homogenous* (single Learning Algorithm) and *Heterogeneous* (varied learning algorithms). In this study, we will go into the details of Homogenous Ensembles. In Homogenous Ensembles, a single learning algorithm is used on separate training datasets resulting in multiple learned models. Different types manipulations can be carried out on the training dataset in order to ease the process of ensembling. Below, we discuss 2 status-quo techniques for training data manipulations in Ensemble Learning.

Bagging

Bagging or Bootstrap [54] aggregating usually involves an equal weighting scheme during the voting step from each model. Formally, given a training set of size n, m samples of size n are created by drawing n examples from the original data **with replacement.** The results of m resultant models are then combined using simple majority voting. The algorithms that benefit the most from Bagging are unstable learners such as decision trees, whose output can change dramatically when the training data is slightly changed [55]. Algorithm 7 describes the basic algorithm for training and testing using Bagging.

Algorithm 7: Bagging: Basic Algorithm [56]

```
Training for t = 1 → T iterations do
    Randomly sample with replacement N samples from the training set
    Train a chosen "base model" on the samples
Testing for m = 1 → M testing samples do
    Start all trained based models
    Predict by combining results from all trained T models: -Regression:
    averaging -Classificaiton: majority voting
```

Figure 2.9 below explains how a Simple Majority voting can help in reaching a good final decision from 3 different classifiers: H_1, H_2 *and* H_3.

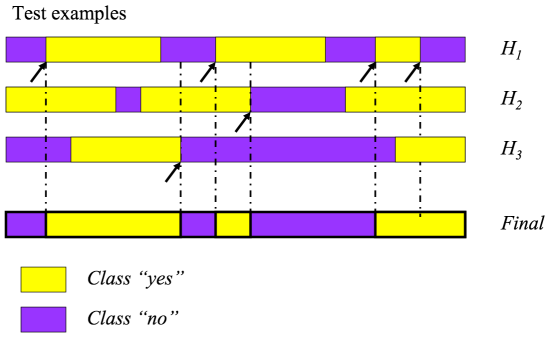


Figure 2.9: Simple Majority Voting [56]

Boosting

Boosting helps to guarantee performance improvements for weak learners, i.e. the learners that only need to produce better accuracy than random guessing (>0.5). A revised implementation for boosting called *Adaboost*, empirically improves generalization performance by building ensembles [57]. In Adaboost, initially all the examples are given uniform weights. At each iteration, with a new-learned hypothesis, the examples are reweighted to bring the focus only on wrongly classified samples. During testing, each hypothesis gets a weight proportional to their accuracy on the training data. Howeer, having weak classifiers can help in strongly avoiding overfitting. [55]

Algorithm 8: Boosting: Basic Algorithm [58]

```
Training
Set all examples to have equal uniform weights for t = 1 → T do
    Learn a hypothesis, h_t, from the weighted examples
    Decrease the weights of examples that h_t classifies correctly
Testing for m = 1 → M testing samples do
    Start all trained based models
    Predict by combining results from all trained T models
```

On average, Boosting produces a larger increase in accuracy as compared to Bagging, although it is not as reliable as bagging. On some occasions, boosting can degrade the overall accuracy while bagging provides a reliable modest improvement. This can be due to the fact that boosting can overfit when there is a significant noise in the dataset [57].

2.3 Related Work

In this section, we will provide a short survey of the available literature. We start by summarizing the available work on Scalable machine learning followed by the work available on our problem. Following this, we provide the literature overview of the existing solutions. All the mentioned websites were accessed in August 2014.

Scalable Machine Learning. As discussed in section 2.2, the process of machine learning essentially depends on three things: the dataset, its feature set and the learning model. If we look back at the recent history, machine learning has gained more popularity due to the increased size of the dataset [58, 59]. Scaling learning algorithms to large datasets is an area with an active research interest [60, 61, 62, 63, 64, 5, 65, 66]. A lot of learning algorithms have successfully been implemented using MapReduce framework. An excellent summary about how some of the traditional learning algorithms such as KMeans, logistic regression, expectation maximization and support vector machines can be efficiently solved in the MapReduce framework is provided by Chu et al. in their study [6]. However, instead of using commodity hardware they used a shared-memory multiprocessor architecture. Through [67, 68, 69], we know that simpler models outperform complex ones trained on smaller data. *But with the increased data comes great computational cost.*

Problem Statement. A lot of efforts [60, 61, 62, 63, 64, 5, 65, 66] have also been spent on solving the problems of learning with large Datasets. Many of the Machine Learning algorithms such as Expectation Maximization and Gradient-based approaches are iterative in nature. In the field of Machine Learning, they are often referred to as *batch* learning algorithms; in a sense that they need an entire "batch" of training samples before updating the model parameter. Studies [70, 8] suggest that the MapReduce framework is unfit for such iterative algorithms. In general, most optimization problems with large feature set and constraint size are not well suited for a realization at scale over MapReduce, if we restrict MapReduce to Hadoop MapReduce. This is mainly due to- the lack of long-lived MapReduce jobs and lack of in-memory computing support. However, a lot of other MapReduce realizations have been introduced. For example, Twister [71] and HaLoop [72] that overcome the iteration based limitation of MR framework and provide in-memory cache as well long-lived MR jobs. These can be applied as possible replacements for performing optimizations on Hadoop MR. But, fault-tolerance is an open issue in both these solutions. Currently, fault-tolerance provided by Hadoop MR outperforms all the other MR based platforms.

A fundamental reason for the limitation of MapReduce is because it causes data to materialize after each iteration, and this stored data is replicated across machines causing substantial disk and network I/O after each iteration. Some major alternative models provide superior performance by alleviating the iteration-based limitation of MapReduce.

Alternative Platforms. *Spark* [13] allows for in-memory cluster computing that allows loading data in cluster's memory and querying it repeatedly, hence addressing the iteration-based limitation of MapReduce. It merges the iterations and materializes the data only when it is required, making it suitable for iterative machine learning/ optimization algorithms. Microsoft's *Naiad* [12]is aimed at supporting incremental iterative dataflows using a new computational model of differential dataflow. It is based on processing the differences between collections but here instead of evolving in one direction of either time or iteration, the collections can evolve in both of them. *GraphLab* [73], specialized for parallel machine learning, allows graphical operations expressing computational dependencies of the data by a framework that allows

asynchronous iterative computations. Hyracks[74] is a partitioned-parallel software platform designed to run data-intensive computations on shared-nothing cluster of computers. It allows users to express computations in the form of a DAG of data operators and connectors and provides a better support for scheduling.

Alternative Algorithms. However, another point of view is suggested by J. Lin, who in his study [14] outlines these limitations and proposes to exploit non-iterative variants instead of entirely giving up on Hadoop. Online Learning techniques have become increasing relevant for Big data environments [75]. Although limited by disk I/O such learners have become quite popular in cases where faster convergence is required. An attractive alternative to training them is to run independent training algorithms in parallel, but on different partitions of data and then combine the final solution. Dredze et al. [70] show that this final solution is however inferior to the output of training the algorithm on the entire dataset.

Proposed Technique. The closest to our approach are the optimizations approaches based on parallelized gradient computation by Nash et al. [76] and a study by Agarwal et al. [65] where they came up with a mixture of online and batch techniques to come up with the most scalable and efficient linear learning system (as of 2011). Their approach was proven to yield a combination of accurate predictions and short training times. However, they try to target the iteration-based limitations of hadoop by proposing an architectural extension while we try to solve the problem at an algorithmic level. Another interesting study is done by Peng et al. [77], where they analyzed the performance of three different optimization strategies to train a Logistic Regression classifier on Hadoop and Spark. They approached the problem in a different way, where they proposed learning by using sublinear methods that optimize the feature-set size on every iteration.

Despite the growing interest in parallelizing large scale learning, there are relatively fewer papers presenting how an end to end solution consisting of machine learning workflows and their end to end integration with the data management platforms is carried out

in enterprises. J. Lin presents a holistic study [15] about the entire large-scale machine learning architecture at *Twitter*. *Facebook* is well known for building its data analytics platform around Hive [23] but still not much is known about its actual machine learning framework. Similar is the case with *LinkedIn* [78] where we know that Hadoop stack is used for a variety of offline and online data processing, but machine learning workflows at LinkedIn are still unknown. *Google*'s anti-abuse efforts for advertisements are summarized in a study [79] by Sculley et al.

Chapter 3

Preliminary Considerations and Proposed Technique

An *optimization* problem is usually used to select the best element from available alternatives based on the given set of criteria. They can be directly used in the field of machine learning to minimize or maximize a particular cost function based on a set of constraints.[1] The function that we need to minimize is usually referred to as the *cost function*. A solution that minimizes the cost function is called an *optimal solution*. For further details on optimization, please refer [80].

Sometimes, the problem can have many "good" solutions referred to as local minima and a "best" solution usually referred to as the "global minima." In such cases, the optimizers usually get stuck in the local minima of the problem, meaning that they do not minimize the function optimally. Although, there are various techniques such as Hill Climbing and Simulated Annealing that start with an arbitrary solution to the problem and iteratively try to find a better solution. More on these in [81]. Another approach is to choose a cost function that is *convex*. According to the convexity property, any local minimum is also a global minimum. This eases the process of optimization as whenever a local minimum exists; it is sure to be the most optimal solution. Interested reader can refer [82].

In section 2.2.3, we decided to use a cost function based on Maximum Likelihood Estimation because its convex in nature, so all the properties of convex functions hold.

[1] We focus on minimization problem but maximization problem can be formulated, similarly.

This helps us from getting stuck at a *local minima*. We further saw that to minimize our quadratic optimization equation (2.5), we can use two different methods of *Gradient Descent* and *Stochastic Gradient Descent*. Both these approaches exhibit different properties as seen in 2.2. In our experiments, we are interested in investigating the difference in performance of three such training strategies for the platforms under consideration: batch/ global, local and hybrid.[2] The difference is measured in terms of how fast the learners minimize the training objective as well as the test error.

Existing Approach. *Global* learning method, usually touted as a better form of learning, keeps the system weights constant while computing the error associated with each sample in the input. The *local* technique, on the other hand, is continually updating its weights, therefore, during the error calculation and gradient estimation phase, different weights are used for each input sample. Hence, although both the algorithms follow different paths during adaption but, theoretically, both converge to the same minimum. For a given dataset, online methods do weight updates for each sample, while batch methods do it for the entire dataset. In general, the batch algorithm is considered to be slightly more efficient in terms of number of computations.

There are various tradeoffs involved in using online and global learning strategies. For example, one of the biggest advantages of *online* learning algorithms is that they optimize the target objective to a rough precision fairly fast. However, due to an inherent sequential nature of these algorithms, the process of parallelizing becomes very tricky. For instance, in Mahout project itself, the implementation of stochastic gradient descent is not parallelized. A good strategy to use them for larger datasets is based on the convex technique proposed by [64, 83]. In their approach, they studied how running an online learning algorithm on each node followed by averaging of their results can be a rather useful technique for obtaining a decent solution. Approximation of results by online training techniques is the fundamental reason for their low adoption.

On the other hand, *Global* learning algorithms such as Gradient Descent are great at optimizing the objective to a high accuracy, once they are in a good neighborhood of the optimal solution. But in general, the algorithms can be quite slow in reaching

[2]Throughout this study, the terms batch-global & local-online will be used interchangeably as they define the same concept.

this good neighborhood. The process of parallelizing global training techniques is fairly straightforward. After each iteration, partial solutions from all the nodes are averaged, first, to be sent as an input to the following iteration, until convergence.

Proposed Approach. In *Hybrid*, the first step involves each node making a single online pass over its local data according to the training technique under consideration (streaming one-pass KMeans or Stochastic Gradient descent). Since, such an online step occurs complete asynchronously without any communication between the nodes, we can use it to quickly get into the good neighborhood of the optimal solution. We can use the obtained suboptimal solution, which is *model* (weight) vector for logistic regression and *centers* vector for KMeans, for initialization of the standard Global Training step. Having an online step before the global step, gives us a good ***warmstart*** for the global step [65]. The key idea behind this is to reduce the approximation that occurred due to the initial online pass.

Irrespective of the type of algorithm, a hybrid training technique usually follows the following processing structure:

1. Pass through the entire local portion of the dataset using the chosen suboptimal training technique.
2. Accumulate the result as a vector of size d (usually dense and of the size of the training parameter)
3. Perform averaging operation on the vector followed by some additional fine-tuning, if required.

An important point to note here is that the local dataset will be orders of magnitude larger than the resultant vector; hence, the number and size of communication operations will remain relatively small throughout this process. Overall, the algorithm benefits from the fast initial reduction of error provided by an online algorithm and a faster convergence in a good neighborhood guaranteed by Global training techniques.

Discussion. We know that the Hadoop-based stack has been around for a while and has been successfully adopted by various organizations such as Facebook, LinkedIn and

Twitter for their traditional data warehousing and business intelligence tasks. But currently, iterative algorithms in Hadoop are a lot slower than they have to be, which leads to comparatively less adoption of Hadoop-based approaches for traditional machine Learning tasks.

The relatively new alternative platforms, discussed in section 2.3, leverage in-memory approach or native support for iterations and flexible data flow take about half to one-fourth of the resources taken by the hadoop that can be attributed to all that avoidable overaload related to serialization and network transfer. GraphLab claims to be about two times faster [73] and hadoop is definitely slower than in-memory based approaches.

If we decide to stick with Hadoop, then there could be many possible implementation level tweaks such as trying to put the smaller half of the join in memory, controlling custom partitioning, not doing random access to HBase and so on. Instead of using commodity machines, we can also shift to a different high-speed rack, such as Amazon EMR [84]. Suggestions to introduce algorithms that can remove the iteration based bottleneck by trading off some accuracy with performance have successfully been adopted to perform large-scale machine learning at Twitter [15] .

An important argument here could be that why Hadoop and its successors are still interesting, even when they have proved to be relatively slower in performance in comparison to the new platforms? The reason might be that the cost of computation is so cheap that being 2 to 4 times slower is not a big deal when compared with the costs and efforts involved in shifting to a new framework or even a new high-speed rack.

Let's assume, we measure the cost of computation in the amount spent per day for the overall computation needs. An enterprise wouldn't mind in spending $2-4X/day for its computational needs when it could be solved using $X on the alternate platforms. What would matter is if it works and uses the infrastructure where the enterprise's legacy data already is and that it scales reasonably without needing a new consultant to shift everything to a new framework. What can actually hurt is spending about $200X/day. This is a likely scenario in various predictive analytics applications owing to the inherent limitation for iteration support in hadoop.

As per our discussion, for predictive analytics, we can achieve this by trading off speed with accuracy using our local training techniques. From our experimental results

in section 5.3, we will see that the local training techniques are up to 10X faster than the global ones. This can bring the cost down to $20/day which wont hurt us but can lead to a non-acceptable level of prediction quality for some scenarios and can beat the overall purpose of predictive analytics. From our experiments, we will further observe that even after applying optimizations in terms of ensembling, our online techniques fail to achieve the level of quality that can be achieved using the Global techniques. In such cases, our proposed approach can help by benefiting both the cost and the overall quality, thereby, keeping the administrators, database experts and data scientists happy.

Our initial set of experiments proves that the hybrid techniques perform better qualitatively for our selected set of bounds in terms of number of iterations. This means that if we had allowed the experiments to finish for our selected convergence thresholds then hybrid is sure to converge faster. Hence, reducing the overall cost and achieving a quality that is similar to the global methods. We can roughly estimate that, upon convergence if the quality achieved by Global is 10Y and online training can only achieve 7Y, then the hybrid implementation can help in achieving an 8.5Y quality level without compromising much on the computation costs. Through this study, we aim to provide a knob in the hands of the designer to trade-off quality and computation costs, by adjusting the various hyper-parameters associated with online and hybrid implementations. We envision that this thesis could prove to be the first step in this direction.

Chapter 4

Implementation

In this chapter, we describe our proposed framework implemented on Hadoop and Flink. In particular, we discuss various design decisions involved in fitting our training approaches to the well known machine learning tasks of Clustering and Classification. For every algorithm, we discuss its fundamental advantages and the inherent bottlenecks.

4.1 KMeans

This section explains the critical parts of the training techniques implemented for KMeans. We start by implementing the Lloyd's algorithm for the global approach followed by using One-pass Streaming KMeans as our local training technique. Further, we present the various implementation details behind our hybrid approach for KMeans.

4.1.1 Global

In the batch version, we distribute the sequential version (algorithm 1) of KMeans. To parallelize, a single KMeans iteration is subdivided in two steps [4]:

- **Assignment**: every data point is assigned to its nearest center.
- **Re-computation**: Centers are recomputed from the assigned data points.

The above process is carried out until convergence. The data is assumed to be stored in DFS in two files: the first containing the data points in the form of point-id/ location pairs *(pID, pLoc)* and the second on containing the initial cluster centers in the form

of cluster-id/ location *(cID, cLoc)* pairs. Initially, *Random Initialization* is used for generating cluster centers for the first iteration. The basic question involves how to perform the initial random sampling on a large dataset. As random sampling usually involves selecting k centroids from n points with equal probability (without replacement). This can easily happen in memory but for on-disk dataset it proves to be infeasible since it would require an efficient (constant) time access to anywhere on disk. In such cases, the solution is to use a *reservoir sampling* algorithm that is designed to sample points from a stream with equal probability. This is usually done in a single pass [4]. The generated results file is then used as input for the subsequent iterations.

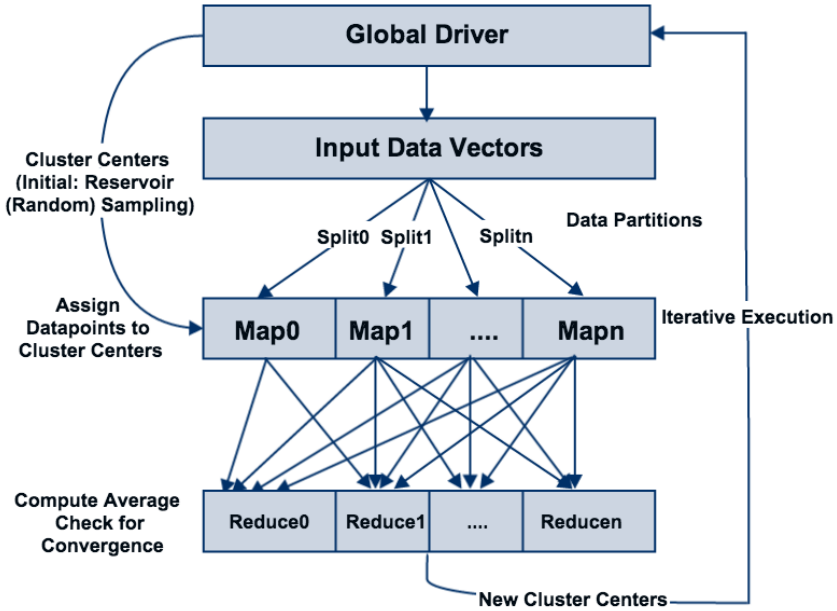


Figure 4.1: MapReduce KMeans: Global Training

Ideally, the entire outer loop should be distributed among mappers and reducers. However, due to the intrinsic data dependency involved, the algorithm a step i cannot proceed without the output centroids from step $i-1$. As can be seen in 9, the inside loop involving a single KMeans step can easily be distributed using MapReduce paradigm.

The *preparation* step usually involves Reservoir Sampling followed by distribution of all the cluster centers to all the mappers in order to help with the *Assignment* step. It

is usually done with the help of ***distributed cache***, where the file containing all the centers is added to it and hence is broadcasted to all the nodes.

For every data point, a mapper computes its pairwise distance to each cluster center. As a result of the assignment step, a *(cID, pLoc)* pair is emitted where *cID* refers to the ID of the nearest cluster center and *pLoc* is the location of the respective point. These points are grouped-by using the *cIDs* and are processed on the same Reducer. Then, the Reducer computes its new location as the centroid from all assigned points. This is usually achieved by computing the geometric average of the pLocs of all the points. The total data transfer volume can be significantly reduced by introducing a Combine function to pre-aggregate, for each center, the centroid on the local machine.

Algorithm 9: KMeans-MapReduce: Basic Algorithm

```
1 Sample k centroids from the n points using Reservoir Sampling
2 while not done do
3     Split the n points across the mappers and for each split, assign the points to
      the k centroids (computing partial sums)
4     Aggregate the partial sums from the k centroids from every mapper into final
      sums and divide by the number of points per cluster
```

The implementation is very robust and is followed in most of the Mahout's clustering algorithms and its pluggable framework allows for the choice between multiple iteration policies (Mapreduce or sequential) and various other use-cases which is outside our domain of consideration. The *stopping condition* is plateauing of the total cost T_C [4].

Discussion. The global training suffers from some inherent drawbacks:

- Due to its *iterative nature*, the global implementation of KMeans consists of chained MapReduce jobs. Depending on the complexity of the problem and the choice of convergence threshold, the algorithm can converge in several iterations. As we have discussed before [14], hadoop jobs suffer from high startup costs that can put a lower bound on iteration times. We also noted that one extra-pass is needed for the sampling of centroids, which is non-trivial and difficult to parallelize, thus adding to the overall cost of computation.

- The *use of random sampling* technique for initialization leads to the invariance in the number of iterations required for the algorithm to converge. In some cases,

the selected centroids maybe particularly unlucky in terms of number of iterations required to converge or can even fail if the initial points are selected from a single cluster. A typical solution of *multiple random restarts* is unfeasible as rerunning the entire algorithm with a new random sample of centroids often leads to an increase in the number of iterations that leads to the escalation of the issue of high startup costs. There is some recent work by [85] in which the authors are trying to come up with an initialization technique that works well with weighted random sampling.

- Iteration $i+1$ cannot start without the input from iteration i, which is the centroid vector in our case. In MapReduce, for fault tolerance, iteration i is first to serialize the result to disk after which it is read by iteration $i+1$. This cost of reading and writing from and to HDFS further adds to the overall computation costs.

- The global KMeans implementation from Mahout doesn't support Fast neighbor search for search that doesn't affect our experiments, but can improve the overall performance of the algorithm.

Flink

The global KMeans implementation of Flink is adopted from the original Flink KMeans program [86]. The code was implemented for one iteration but adding an outer driver program to control the loop on Flink was trivial. The key difference in computation comes from the inbuilt support for Iterations, as discussed in section §2.1.

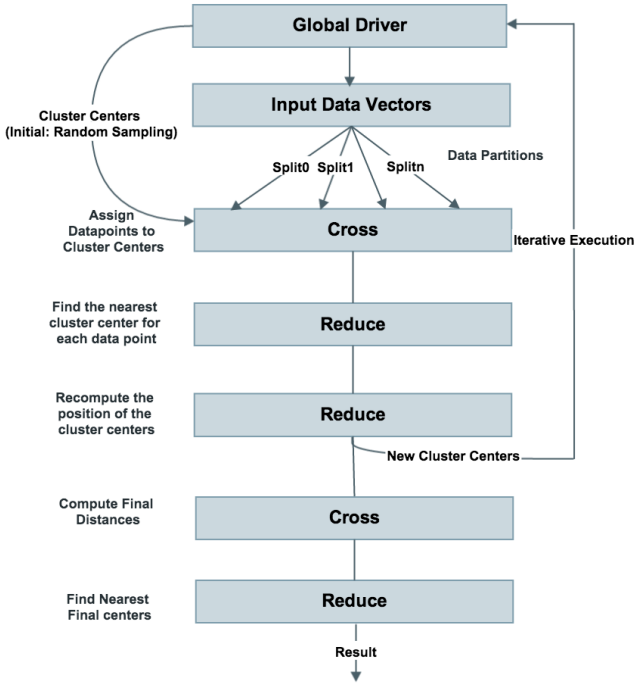


Figure 4.2: Flink Global KMeans Architecture, adapted from [86]

4.1.2 Local

In many cases, the amount of information is extremely big and hence we cannot afford to store the entire information or even processing the same object twice. In those cases, we make use of streaming algorithms, which store a *sketch* of the input data and produces an approximate answer. For the local/ online implementation of KMeans, we follow the algorithm proposed by Schindler et al. [87].

Algorithm 10: KMeans: Local Training - Basic Intuition [4]

```
for each point p with weight w do
    Find the closest centroid c to p and let d = dist(c, p)
    if even with probability proportional to d × w/δ occurs then
        create a new cluster with p as its centroid
    else
        merge p with c updating c
    if there are more than O(klogn) clusters then
        δ ← β × δ
        collapse the clusters recursively
```

In this implementation, unlike general KMeans, the number of clusters to generate is not fixed as k and are allowed to vary and is on the order of $O(klogn)$. This allows for easily maintaining the invariance. The algorithm follows 3 major steps:

- *one pass* over all the points selecting those that are far away from the ones that have already been selected.
- a *re-clustering* step to dismiss those centroids that are less interesting
- *Weighted* points are then clustered in-memory using a traditional *Ball KMeans* approach.

The approach is summarized in algorithm 11.

Algorithm 11: KMeans: Local Training

```
Initialize f = 1/(k(1 + logn)) and an empty set K
while some portion of the stream remains unread do
    while K and some portion of the stream is unread do
        Read the next point x from the stream
        Measure
        if probability then
            set K ← K
        else
            assign x to its closest facility in K
    if stream not exhausted then
        while K > k do
            Set f ← f
            Move each x to the center-of-mass of its points
            Let w_x be the number of points assigned to xK
            Initialize K containing the first facility from K
            for each x ∈ K do
                Measure
                if probability event occurs then
                    set K ← K
                else
                    assign x to its closest facility in K
            Set K ← K
    else
        Run batch KMeans algorithm on weighted points K
        Perform ball KMeans on the resulting set of clusters
```

The map-reduce implementation of streaming KMeans is fairly straightforward. Multiple mappers can produce the independent *sketches* from their chunk of dataset that can later be merged together on a single reducer to produce the final set of centroids.

First, the initial cut-off is computed after which the input data is split, and each split is passed to the mapper that consists of an instance of Streaming KMeans algorithm as shown in 11. The results from each split are then passed to a single reducer for the final clustering step.

At the *reducer*, the merged results consist of a set of C centroids that is much larger than the final desired number of clusters k but should be small enough to fit in the memory. These C centroids are weighted according to the number of points assigned

to them during the initial one-pass streaming phase. Since C is relatively small, this final clustering step is very fast compared to the cost of clustering the original data. In order to get the resultant k centroids, a 2-step clustering process is carried out at the reducer. Initially, a *KMeans++ initialization* is carried out to select k centers out of C. As a result, these centers have considerably good diversity without being subject to the pathological behavior due to outliers. Finally, *Ball k-means* algorithm (refer algorithm 4) is used to perform the final tune-up. It adjusts each of the final k centers by recomputing the final cluster centroids based only on the close members of the cluster rather than all its members.

Discussion. An important point to note here is that the size of approximation is $klogn$ where $klogn << n$ where n refers to the total number of input data points. Hence, we can argue that the resultant data is expected to fit into the memory. Now, as the entire data is available in memory, all the known tricks of Global KMeans training can be applied to further optimize the results. Hence, KMeans++ initialization and multiple restarts, as discussed in the former sections, are all possible [4].

As we will later see in our experiments, that even after such optimizations, the quality of clustering achieved through local training will be close to the actual solution but still not optimal. This could be attributed to the fact that the algorithm uses carious approximation techniques.

Also, since only a single reducer is used to perform all the final computations, this often leads to the creation of ***stragglers*** in the reduce phase. As discussed in Chapter 2, in this situation, a single reducer can often get overloaded due to its limited resources in relation to the overall computation task and can often become counterproductive in terms of performance.

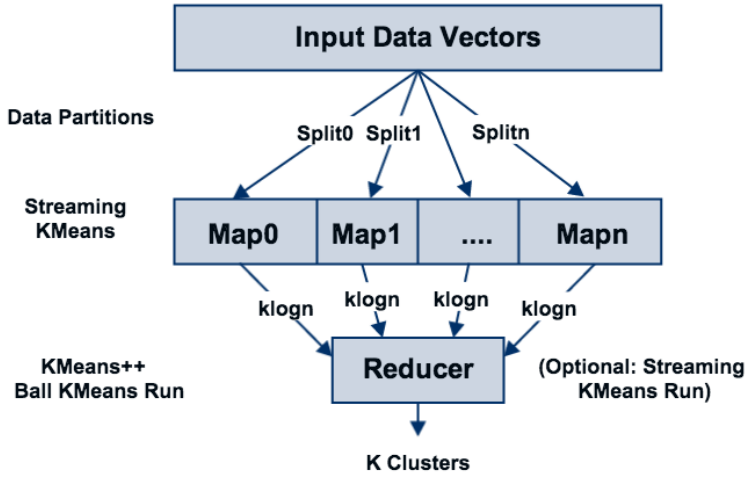


Figure 4.3: MapReduce KMeans: Local Training

Flink

In our implementation of Streaming KMeans, we tried to reuse the basic classes from Mahout. The architecture followed is similar to the one using MapReduce. We tried avoiding the use of *Cross* operator.

4.1.3 Hybrid

As we have seen, the unavoidable random-initialization often adds to our woes related to the hadoop startup costs as the number of iterations until convergence depends on the choice of initial centroids. It can even fail if two initial centroids are selected from the same cluster. But we know, that the faster local-training implementation often leads us to a good enough solution, whereby, we can be sure that we are near the actual solution. This can lead to a decrease in the total number of iterations and can help us avoid the situation of failure of the entire job. Thus, the hybrid version of the algorithm targets to get the "best of both worlds" by using the approximate, but good quality centroids from local-training to get to the optimal solution faster. Figure 4.4 summarizes the entire solution-

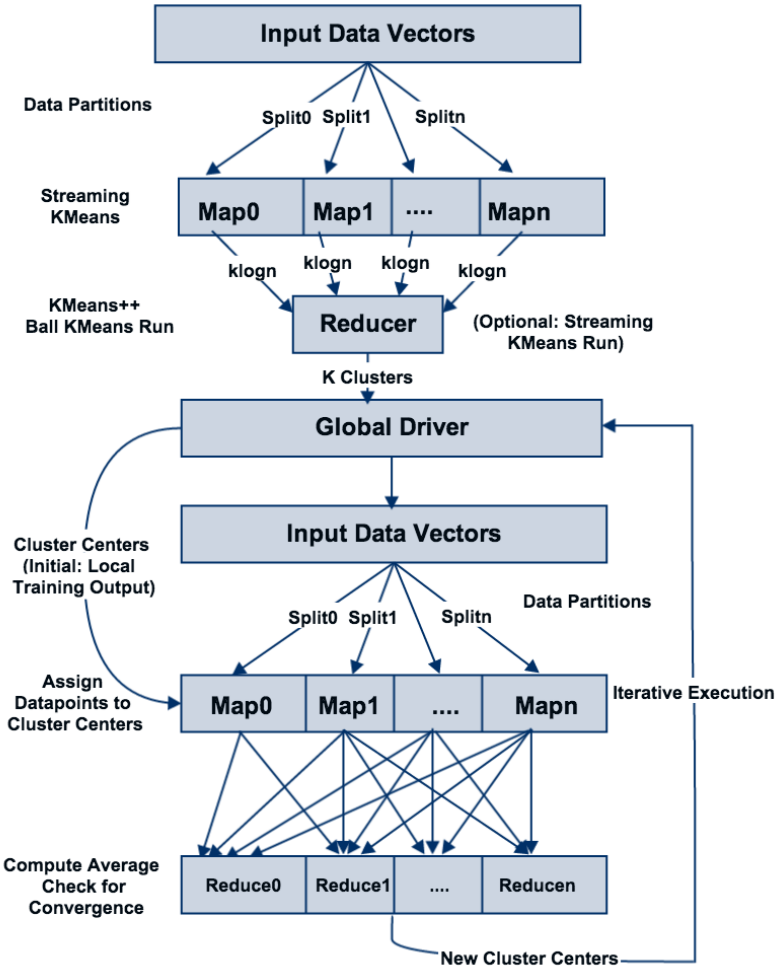


Figure 4.4: MapReduce KMeans: Hybrid Training

The MapReduce implementation is pretty intuitive where the local training job can directly be linked to the global job in a way that the output of the local algorithm becomes the input of the global one. A possible trade-off could be the serialization and de-serialization of the model parameter to and from HDFS from local to the global training. In our experiments, we discuss this possible trade-off that whether the gain in performance and quality overpowers the startup cost+ serialization/deserialization cost+ cost of running an extra local algorithm. The results are meant to be used by the possible designers to chose between the options available keeping the application requirements in mind.

Flink:

The Hybrid Implementation is quite similar to that of MapReduce. We try to use the output of local training as an input for the global one. However, the hybrid implementation benefits from the ability of the platform to support custom flexible dataflows. Hence, unlike MapReduce, in Flink, there is no need to run two separate jobs and the serialization and subsequent deserialization of the results from local and global training, respectively. The output of the local can directly be inputted to the global training for the needed "warm-start."

4.2 Logistic Regression

In this section, we cover the details behind the implementation of Logistic Regression. Further, we discuss the effects of the existing optimization strategies of Batch Gradient Descent and Stochastic Gradient Descent in fitting the learning model to our hypothesis. We start with the Global training followed by the discussing the intricacies behind the local training. We finish by discussing how a proper amalgamation of the local and global can lead to our proposed hybrid training approach.

4.2.1 Global

As we have seen in figure 2.7, the loss function and thus, the gradient decompose linearly thereby, making MapReduce implementation of Gradient Descent fairly straightforward [14]. A MapReduce friendly implementation of Gradient Descent is also discussed in [6]. In that, they rely on distributed computation of gradients locally on each computer that holds parts of data and subsequent aggregation of gradients to perform a global update step.

As can be seen in figure 4.5, on each mapper, we process each training example in parallel and compute its partial contribution to the gradient. These are then emitted as an intermediate key-value pair and shuffled to a single reducer. The reducer then sums up all these partial gradient contributions and updates the model parameters (weights). This entire process corresponds to a single iteration in Gradient Descent algorithm and usually takes a single MapReduce job. The complete training usually requires a lot of

synchronization sweeps that are inherently several MapReduce iterations chained in a sequence. The number of these iterations usually depends on the overall complexity of the problem. The iterations are usually handled by the Global Driver program that sets up the job, waits for it to finish by checking for convergence (refer section 2.2.3.1) and repeating as long as necessary.

In figure 4.5, the mappers computing the partial gradients with respect to the training data require an all-time access to the current model parameters (weights). In general, this can be done by loading them as "side data" [14] in each mapper, which can be done in Hadoop either by loading directly from HDFS or by using distributed-cache. Without this, there would be no other way to perform multiple iterations. In our implementation, we use distributed-cache for such intermediate storage. Since, the model parameters are updated at the end of each iteration, the updated model is required to be passed to the mappers at the next iteration. Introduction of combiners to perform the partial aggregation or in-mapper combing pattern can help in further optimization [58].

Discussion. As discussed in [6], the algorithm scales linearly with the amount of data and log-linearly in the number of computers. However, this implementation suffers from several drawbacks-

- Hadoop jobs suffer from having high start up costs. It can be even tens of seconds [14] on a large cluster under load. This usually places a lower bound on iteration times. As discussed above, depending on the complexity of the problem, the global training can reach convergence after several jobs, thereby suffering from the unavoidable drawback of startup costs each time it starts an iteration.

- As can be observed, the implementation suffers from the drawback of overloading a single reducer to perform all the computations. It often creates stragglers in the reduce phase. It is often caused by data-skew but is an intentional implementation strategy for our case and cannot be avoided by speculative execution.

- In order to perform the final aggregation, the reducer must wait for all the partial gradients thereby waiting for all the mappers to finish. Hence, the speed of each iteration is bound by the *slowest* mapper.

- The model parameter is serialized to disk at each iteration before getting loaded again to the distributed cache for the next iteration. Even though, this provides excellent fault tolerance but it comes at the cost of performance.

- This combination of a single reducer and stragglers leads to poor cluster utilization and thereby affecting the final throughput.[1]

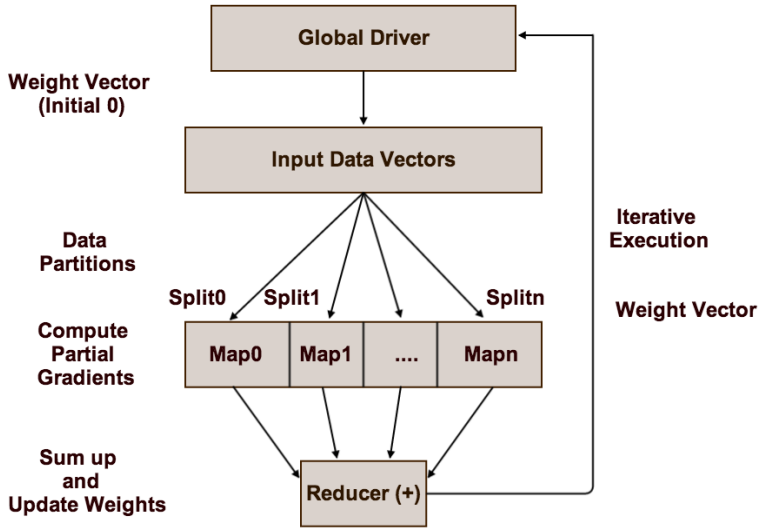


Figure 4.5: MapReduce Logistic Regression: Global Training

Flink

As can be seen in figure 4.6, the overall architecture of Global training is quite similar to that of MapReduce. Instead of Mappers, cross operators are used to overcome the lack of distributed cache based support in the last stable release of Flink. [2] This adds a little overhead in comparison to a faster MapReduce implementation, however, inbuilt support for iterations by Flink helps in overcoming iteration-based drawbacks of MapReduce.

[1] This situation can be avoided by running other jobs on the cluster at the same time but for experimental purpose this is not an ideal situation.

[2] Flink now supports Distributed cache both at API level and in terms of Broadcast variables.

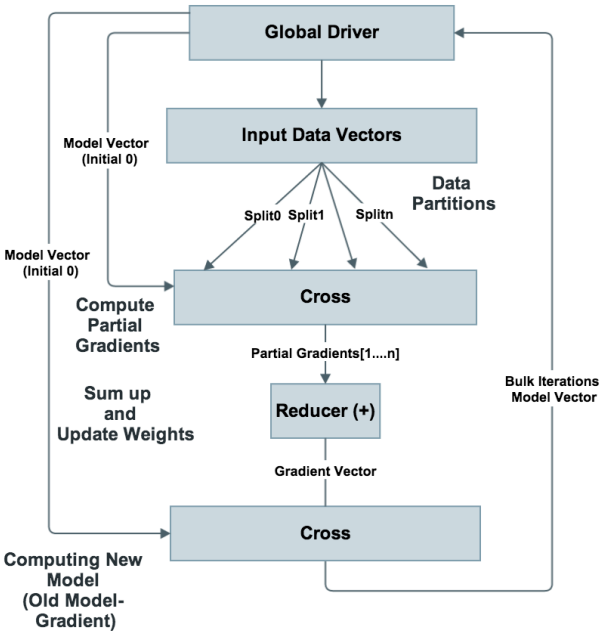


Figure 4.6: Global Training Architecture: Flink

4.2.2 Local

As discussed above, one of the major shortcomings of gradient-descent training is the use of iterations, then why not simply get rid of it. [14] proposes a new approach in which instead of running batch gradient descent to train classifiers, we can adopt a local training technique of stochastic gradient descent. The basic idea is that instead of updating the final model after considering all the training samples, why not update the model after each training example.

Algorithm 12: Logistic Regression: Local Training

Define $T = \lfloor m/k \rfloor$
Randomly partition the exammples, hence each machine receives T examples
forall the $i \in \{1, ...k\}$ **parallel do**
    Randomly shuffle the data on machine i
    Initialize $w_{i,0} = 0$
    forall the $t \in \{1, .., T\}$ **do**
        Get the tth example on the ith machine (this machine), $c^{i,t}$
        $w_{i,t} \leftarrow w_{i,t-1} - \eta d_w c^i(w_{i,t-1})$
Aggregate from all computers $v = \frac{1}{k}\Sigma_{i=1}^{k} w_{i,t}$ *and* **return** v

[5] introduces a data parallel stochastic gradient technique that enjoys a number of key properties that make it highly suitable for parallel, large-scale machine learning. It imposes very little I/O overhead as the training data is accessed locally, and only the model is communicated at the very end.

Although, this approach addresses the iteration bottleneck, but it still doesn't solve the single reducer problem. For that, we can leverage the strength of ensemble-based training [88, 89]. As discussed in [15], instead of training a single classifier, we can train an *ensemble* of classifiers and combine the predictions of each using the techniques of majority voting and weighted interpolation. Training each classifier on a partition of training examples and then using ensemble-based training to combine the results has been proved to be very effective for parallel environments [63, 64]. This lets us control how the learning is carried out. Hence, by controlling the number of reducers we can control the number of models that will be learned.

In our implementation, this is achieved by choosing the *number of partitions* initially. So if we set the number of partitions to 1 then all the training instances will be shuffled to a single reducer and hence, fed to only a single learner. Upon setting the *number of partitions* to $n > 1$, the training data is split $n - ways$ and n different models are independently trained on each partition. This allows us to scale-out easily. As discussed in section 2.2.3.3, an ensemble of classifiers trained on partitions of large datasets outperforms a single classifier trained on the entire dataset as it lowers the variance in error [15].

In case of online trainings, the learning models are usually dependent on the order of samples, this can be accomplished by generating random numbers for every training

instance and then sorting them, which can lead to an extra MapReduce job. In our implementation, we achieve this by using random numbers to send the training instances to random partitions. However, this can be improved by following the former strategy. The entire process can be summarized in the figure 4.7.

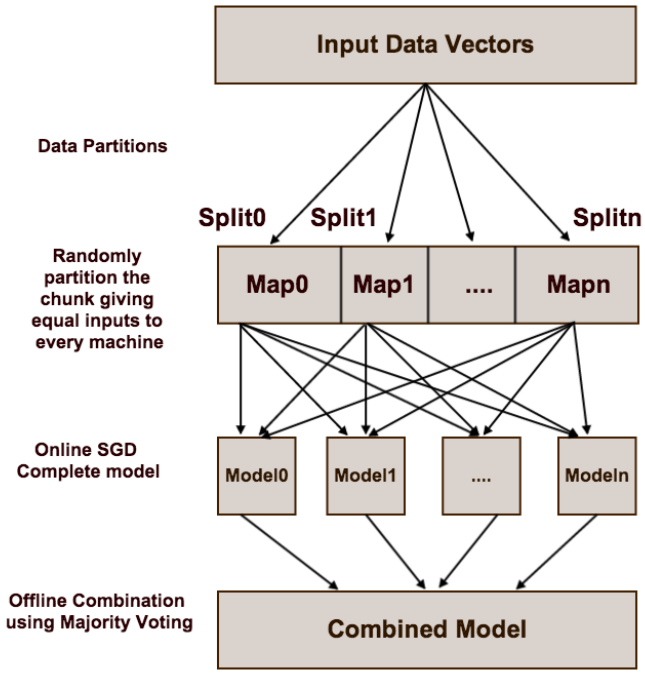


Figure 4.7: MapReduce Logistic Regression: Local Training

As can be seen in section 5.3, in most cases, tuning the ensemble size can help in increasing the qualitative performance for Local Training in Logistic Regression. Although, the local training is faster as it removes all the major performance bottlenecks in Global training, but its online nature only leads it to an approximate solution which in most cases is in the good neighborhood of the actual solution.

Flink

Flink implementation for local training follows the same flow as the one on MapReduce where Mappers are used for randomly partitioning the input data to different reducers (read: partitions) where Stochastic Gradient descent based local training is carried out,

generating a number of models. As we know, in MapReduce, another job was needed to perform ensembling of these results (that can be done offline). In Flink, ensembling of results is carried out by just adding another reduce in the plan (DAG). In our experiments, however, we limit the local training only until the model training phase but such an optimization is helpful in reducing the computation time for Validation phase.

4.2.3 Hybrid

As discussed above, both the local and global training techniques have their set of pros and cons and suffer from an obvious trade-off between quality and performance. Continuing our discussion from chapter 1 and chapter 3, let's assume a company that has successfully adopted MapReduce framework due to the growing popularity of Hadoop-based stack at the data processing framework of choice. The company consists of both database experts and data-scientists. Database experts would give more weight to having faster results over the best ones while the case will be completely opposite from the point of view of data scientists.

We know that the local training using stochastic gradient descent mostly comes up with a good-solution that is usually in the neighborhood of the optimal one. From the discussion in section 2.2.3.1, we know that the global training using gradient descent can quickly reach the optimal solution once it is in its neighborhood. Using the proposed appraoch from 3, we propose to give a "warm-start" to global training by adding an online step to the overall algorithm. Our approach is summarized in figure 4.8.

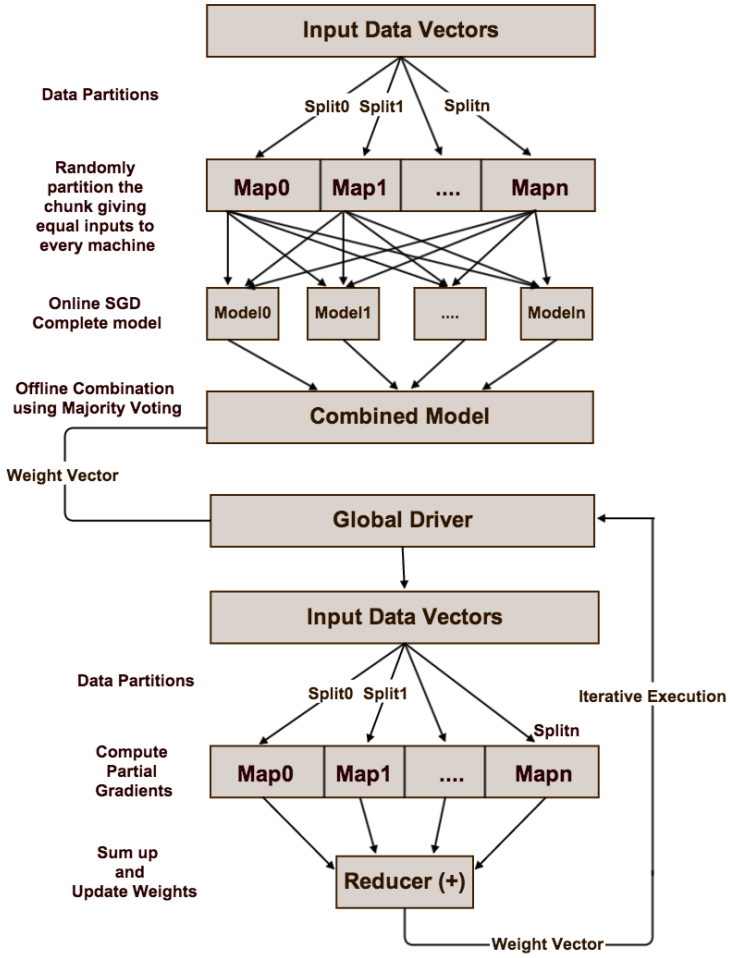


Figure 4.8: MapReduce Logistic Regression: Hybrid Training

Intuitively, the hybrid training should perform at least as good as the local one. Adding a step of global-training should only add to the quality achieved by the local one. Also, since its in the neighborhood of the optimal solution, it should reach the optimal solution in less time, thereby, fewer iterations. We further test our claims in section 5.3.

The MapReduce implementation for the hybrid technique is fairly straight-forward. It can easily be achieved by chaining the local and global training jobs. An additional overhead comes from the cost involved in the communication of the model parameter (weights) which has to be re-read from the disk after local-training. However, the size of a single model is much less than the ßsize of the dataset and usually equals the size of

a single data point. Job start-up times also add to the overhead, but they are typically unavoidable in our case.

Flink

Just like the hybrid clustering implementation, the logistic regression implementation also benefits from flexible dataflow support for large DAGs. It allows to link the output of local training directly to the global implementation without the need to stop the local training and start another job and without the need to write the local output to HDFS to be read by the Global trainer again. The architectural semantics are quite similar to the one in MapReduce.

Chapter 5

Experimental Evaluation

It is organized as follows. In the first section, we explain the underlying hardware and software environment which is followed by the description of the high-dimensional data sets. In the end, we present the experiments performed and their results.

5.1 Experimental Setup

Hardware Environment

All experiments were executed on a cluster of 8 identical machines and one additional machine as the master. Each machine has two IBM 8231-E2B CPUs with 4 cores each, 48 GB of RAM and 4 separate disks with 600 GB each, connected with gigabit ethernet.

Software Versions

For the purpose of the report, we have used Hadoop version 1.2.1.[1] For Flink, we have used 0.4 stable release which was the most-stable release available at the time of implementation.[2] Both Flink and Hadoop were executed on Java version 1.7.0 from Oracle on IBM J9 VM with JIT and AOT enabled. Both the master and slaves run Linux Fedora Server 19.

[1] http://hadoop.apache.org/releases.html#Download, accessed April 2014.

[2] Flink was built from the source code available at https://github.com/apache/incubator-flink/tree/release-0.4, accessed April 2014.

Configuration

Both Hadoop and Flink require individual tuning parameters that can have a direct impact on the overall performance. To maintain lucidity, we won't go into the details of these parameters. Due to some initial performance drawbacks,[3] both the systems were configured to optimally use all the available resources. Although, the optimal settings highly vary for each experiment, we have tried our best to ensure that each and every experiment was executed with the same setting.

Property	Value
CPU Model	IBM 8231-E2B
Number of nodes	10: 1 x Master, 8 x Workers
Heap/ node	25 GB Heap
Number of CPU(s) per node	2 Quad-core
RAM per node	48 GB
Disk per node	4 x 600 GB
Interconnection Method	Gigabyte Ethernet
Operating System	Linux Fedora

Table 5.1: Cluster Information

5.2 Dataset Information

In this section, we present the details of the datasets that we used for clustering and classification. To maintain uniformity and consistency and to encourage ease in experimental-reproduction, we made sure to use a single input format for all our experiments. **SVM-Light** [90] format is a simple sparse-vector encoding format. It is a line-based format, i.e. a line represents a single training instance. As can be seen in figure 5.1, each line begins with either {+1,-1} or {1,0} based on your choice of *class labels*. It is then followed by a list of features separated from it through a space. *Features* are organized as {featureid,

[3]Using wrong JVM (http://openjdk.java.net/projects/zero/) on the cluster, that doesn't use JIT'ing. Therefore, the performance was degraded by up to 10x.

value} pairs with a colon(:) as delimiter. A snapshot from a sample input file is shown in the figure below:

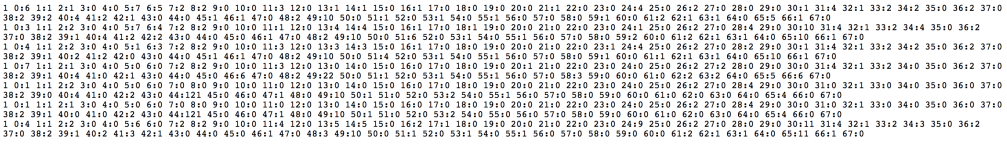


Figure 5.1: SVMLight Format: Example

5.2.1 Clustering

The initial experiments were performed on a 2-dimensional synthetic dataset in which points follow Gaussian curve near uniformly distributed centers. However, since such synthetic datasets are often assumed to be easily clusterable, we also used real-world datasets to obtain the practically relevant results. The code for generation of the synthetic dataset is adopted from [91]. Our main source for the real world dataset were the UCI Machine Learning Repository [92] (*Census 1990)* and a similar cluster-comparison study (*BigCross*, which is a cartesian product of Tower and Covertype datasets from [92] with 11620300 data points at 57 attributes). The size and dimensionality of the datasets are summarized in table 5.2

	# of Data points	# of Dimensions	# of Centroids	Type
Synthetic	1000000	2	40	Float
US Census 1990	2458285	68	-	Int
BigCross	11620300	57	-	Float

Table 5.2: Overview of Clustering Datasets

5.2.2 Classification

For classification experiments, we chose 4 open datasets to run our quality and performance based test programs. The details are shown in table 4. All the mentioned datasets are *sparse* in nature. We split each one of them into training and testing sets.

```
{

"uri":"http://zzzoot.blogspot.com/2008/06/re-reading-godel-
    escher-bach-eternal.html",
"clueweb_docid":"clueweb09-en0027-77-18166",
"sentence":"I have decided to re-read Douglas Hofstadter's Godel
    , Escher, Bach: An Eternal Golden Braid \".",
"entity1":"Douglas Hofstadter",
"freebaseId1":"/m/02fcx",
"freebasetyp1":"/m/people.person",
"entity1_offset_1":26,
"entity1_offset_2":44,
"xml":"...."

}
```

Listing 5.1: Raw Data Format

The **20NewsGroup** dataset is well-known for evaluation of LR model and has a balanced distribution between positive and negative data instances. It covers a use-case of topic classification [77]. The **Gisette** dataset is smaller in the number of datapoints and the features are also less sparse. The datasets are summarized in table 5.3. Preprocessed version of **Reuters Corpus Volume 1(RCV1)** [93] uses a combination of 2 labels for both the positive and the negative labels. For our ensemble-based experiments, we have used **Kdd Cup** [94] dataset which was shown to improve accuracy, during the tournament, by the use of ensembles. To increase the complexity in terms of feature-set size and dimensions and to extend our use-case application to *Named-Entity Recognition*, we used **ClueWeb09** dataset [95] which consists of English-web pages annotated by researchers at Google. The annotation was done using **Freebase ID** [96]. For training, we used a dataset which was preprocessed using *dkpro* [97], for a single "person" entity type, where every data point was an XML java object that looked like the one shown in listing 5.1.

For the purpose of our experiment, we needed to prepare training and testing datasets for the use case of identifying a person from a given sentence. To achieve this, we extracted the sentence and its corresponding person entity using a java parser. We removed possible duplicates by writing a simple MapReduce job that groups-by sentences.

For our task, we decided to use hashed word 3-grams as features. The actual feature extraction phase was carried out on the reducers. Here, the feature extractor treated every datapoint as a raw array of words and moved a 3-word sliding window along

```
public static int createHash(String entity){
int hash=7;

for (int i=0; i < entity.length(); i++) {
 hash = (hash*31+entity.charAt(i))% $desiredfeaturesize;
        }
return hash;
}
```

Listing 5.2: Feature Hashing Algorithm

the array while hashing the contents of the words whose value was later taken as the feature id. Features were treated as binary-based on their presence or absence, i.e. the feature values were always one, even if the data row contained multiple occurrences of the same word. This leads to a large feature vector that is very sparse. Thus, except tokenization and removal of extra symbols to keep only alphanumeric characters, we made no attempt to perform any linguistic processing. The 2^{nd} word from our 3-gram feature size decides the class label, i.e. if the 2^{nd} word belongs to the corresponding entity then it is labeled as positive or vice-versa. This leads to an obvious skew in the number of positive and negative labels where the negative labels outnumber the positive ones by a huge difference. For the purpose of our final tests, we decided to only use a sample consisting of a third of the total number of datapoints. Listing 5.2 shows a typical hash function used by us.

# Name	# of Datapoints	# of Features
Gisette	6000/1000	5000
RCV1	20242 / 677399	47236
20NewsGroup	19996	1355191
ClueWeb	2807257/711920	10000000
Kdda	8407752 / 510302	20216830

Table 5.3: Classification Data Set Description. Two values signify the training and testing set respectively.

5.3 Experimental Results

As our basic ***test methodology***, whenever the size of the dataset allowed it, we executed 5 trials for every experiment and report their average to guarantee the statistical significance. Following the same methodology as one of such similar study [27], we discarded an additional first run just after the system start-up to consider only the "hot-state" of the system, a term coined in the afore-mentioned study. In total, there are 6 test programs for both clustering and classification respectively. They try to compare the 3 different training paradigms as discussed in 3 on the dimensions of Quality and performance. The run-time refers to the wall-clock-time that computes the total time elapsed for running the actual job. So, the time for additional tasks such as system-startup and dataset loading is ignored since it equally influences all the three training techniques equally. The results are summarized in the form of tables and charts, which are self-explanatory, however, in some cases, text has been added to support the values.

5.3.1 Clustering

In this section, we start by discussing the scoring techniques followed by the heuristics employed for the actual experiments. In the end, we finish by presenting all the results, while discussing about some of the interesting ones.

5.3.1.1 Quality Measures

The basic quality measure for all experiments is the sum of squared distances, to be referred to as **costs of clustering**.

To measure the clustering cost, simply compute the sum of squared distances from points to their respective centroids. There is an inherent trade-off to this approach. The way to compute the nearest centroid depends on the implementation of the user. As mentioned in the background section, there are various ways to find the nearest-neighbor. The slowest but the most accurate is Brute Search. But, projection search scales well in higher dimensional large datasets. In our cost-implementation, we have used Projection Search as the chosen technique to find the nearest neighbor.

Dunn-Index

Invented in 1974 by J. Dunn [54], the Dunn Index combines the intra-cluster distance, $\triangle_i$ for cluster i and the inter-cluster distance between cluster i and j, $dist(c_i, c_j)$. $\triangle_i$ can mean different things. For example, for cluster X_i, it could mean:

- the maximum [4] distance between any two points: x and y

$$\triangle_i = max_{x,y \in X} dist(x, y) \tag{5.1}$$

- the mean [4] distance between any two points: x and y

$$\triangle_i = \frac{1}{\mid X_i \mid (\mid X_i \mid -1)} \sum_{x,y \in X_i, x \neq y} dist(x, y) \tag{5.2}$$

- the mean [4] distance between any point and the centroid: x and y respectively

$$\triangle_i = \frac{\sum_{x,y \in X_i, x \neq y} dist(x, y)}{\mid X_i \mid} \tag{5.3}$$

- the median [4] distance between any point and the centroid. This is the one used in the implementation because-
  - a median is much more robust to outliers than mean or max
  - computing the distances between all pairs of points is not feasible for larger datasets.

So, the Dunn Index [98] is defined as

$$min_{1 \leq i \leq k} \left\{ min_{1 \leq j \leq k, j \neq i} \left\{ \frac{dist(c_i, c_j)}{max_{1 \leq l \leq k} \triangle_l} \right\} \right\} \tag{5.4}$$

where $dist(c_i, c_j)$ represents the distance between cluster i and cluster j, and $\triangle_l$ measures the intra-cluster distances of cluster l.

The basic aim of Dunn Index is to identify dense and well-separated clusters. Since, we seek to find clusters with high intra-cluster similarity and low inter-cluster similarity, the clustering algorithms with ***higher*** Dunn Index are desirable.

Davies-Bouldin Index

Invented in 1979 by D. Davies and D. Bouldin [98], the Davies-Bouldin Index is an intracluster evaluation scheme. The Index is defined as [98]:

$$DB = \frac{1}{k}\sum_{i=1}^{k} max_{j\neq i}(\frac{\triangle_i + \triangle_j}{dist(c_i, c_j)}) \tag{5.5}$$

Intuitively, the algorithms that produce clusters with low intra-cluster distance and high inter-cluster distances will produce a low Davies- Bouldin measure. Hence, a ***lower*** Davies-Bouldin Index indicates a better clustering.

Measure	Desirability
Cost	Lower is better
Dunn Index	Higher is better
Davies Bouldin Index	Lower is better

Table 5.4: Overview of Clustering Quality Metrics

5.3.1.2 Experiments and Results

As we know that one of the biggest drawbacks of KMeans algorithm is that, the final clustering varies with the choice of initial k. Finding its global minimum is, hence, an NP-hard problem. Also, Streaming algorithms are known to have a randomized nature; hence, the results shown in this study are averaged over 3 runs (multiple restarts) for each value of k. We usually ran the experiment for 10 iterations for each of the chosen value of k.

Best k	Running Time			Cost		
	Local	Global	Hybrid	Local	Global	Hybrid
Synthetic ($k = 20$)	282	614	838.5	2637.57	2611.25	2723.48
US Census 1990 ($k = 40$)	810	852	1629	4756.23	4909.78	4749.16
BigCross ($k = 10$)	917	663	1637	93068.86	94935.42	92979.56

Best k	Dunn Index			DB Index		
	Local	Global	Hybrid	Local	Global	Hybrid
Synthetic ($k = 20$)	0.18	0.62	0.63	0.98	0.549088	0.48
US Census 1990 ($k = 40$)	0.10	0.11	0.21	3.73	4.418694	2.03
BigCross ($k = 10$)	1.15	1.91	1.99	0.71	0.63	0.53

Table 5.5: Comparison: Experimental Results for the best value of K

From our discussion in chapter 2, Clustering is a type of Unsupervised learning technique where the number of classes is unknown. So, not only their value but also the number of initial k affects the overall clustering. So, all our experiments were conducted for each of the 3 different values of k (10, 20 and 40). We do not assume in anyway that the actual number of clusters should belong to one of the given options. The idea is to capture the trend by checking how the change in quality metrics can help in deciding the best k, keeping in mind the type of the training.

Running Times:

The first set of experiments was carried out to see the difference in overall running times of the three approaches overall different datasets. The results are shown in figure 5.2. The local implementation clearly outperforms the other approaches on all the three datasets. This is mostly because it doesn't make more than one-pass over the entire dataset. This effect could be attributed to the iteration-based overhead associated with global and hybrid approaches. As discussed above, we have set an initial threshold on the global and hybrid training over the number of iterations. The purpose of these experiments is to show the difference in running times of the local approach in comparison to the other training techniques. So, we can successfully observe that local outperforms the other approaches in terms of average runtime performance.

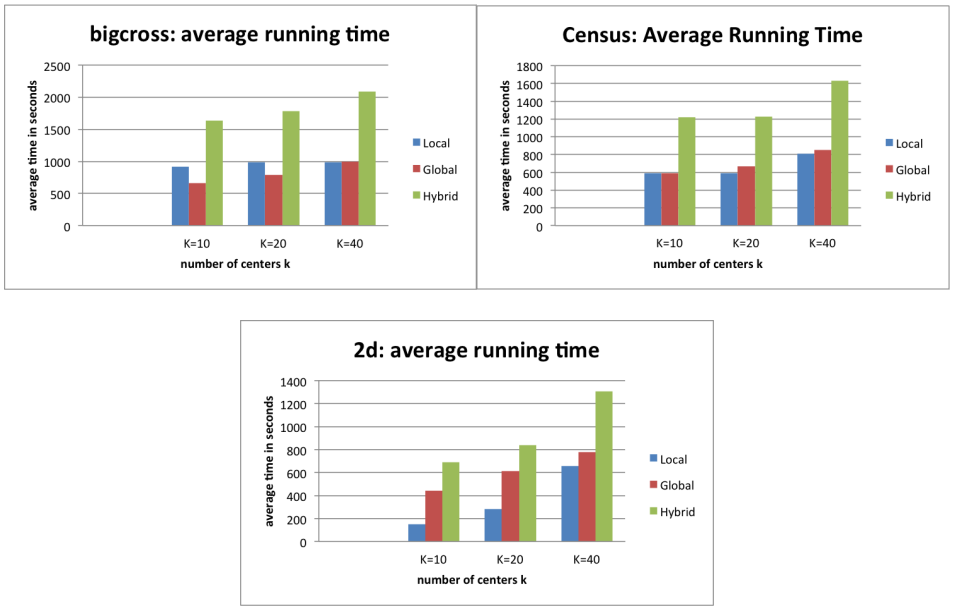


Figure 5.2: Experimental Results: Running Time Performance

As can be guessed, in all the cases, Hybrid implementation takes longer than batch which is nothing unusual since the jobs are still far from convergence. For a limited number of iterations, hybrid takes longer due to an initial online pass followed by the thresholded batch iterations.[4] This can also be attributed to the time spent in serialization and deserialization of the model parameter which is exchanged between the local and global jobs. We believe that due to Flink's native support for iterations and large DAGs such an effect can be further reduced helping in the faster convergence of Hybrid approach.

We further observe that irrespective of the type of training approach, as the number of Centroids increase, the overall running time increases. This could be due to the increased computation for calculating the distance from every point to all the centroids in the mapper.

[4]A fair metric that can help in comparing the running times of global and hybrid approaches could be the total running time till convergence. We propose to perform these as a part of our future work.

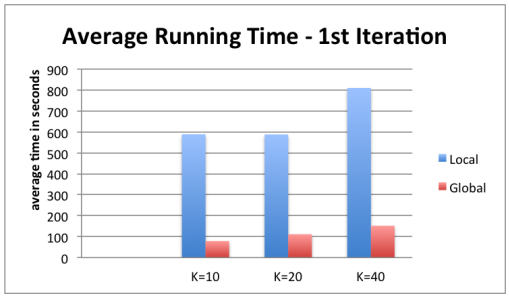


Figure 5.3: Global vs. Local Running times for one Iteration

However, as we note from figure 5.3, if we compare a single iteration time for all the approaches, we observe that one iteration of local usually takes longer than any iteration of the batch implementation. This could be mainly because of:

- the combination of waiting due to a slow mapper and Straggler Effect
- local implementation's computation time gets affected by the choice of different hyperparameters. For example, additional Ball KMeans runs performed to increase the robustness of the results also affect the total computation time. This can also be the reason as to why global training beats local on smaller number of centroids.

Quality:

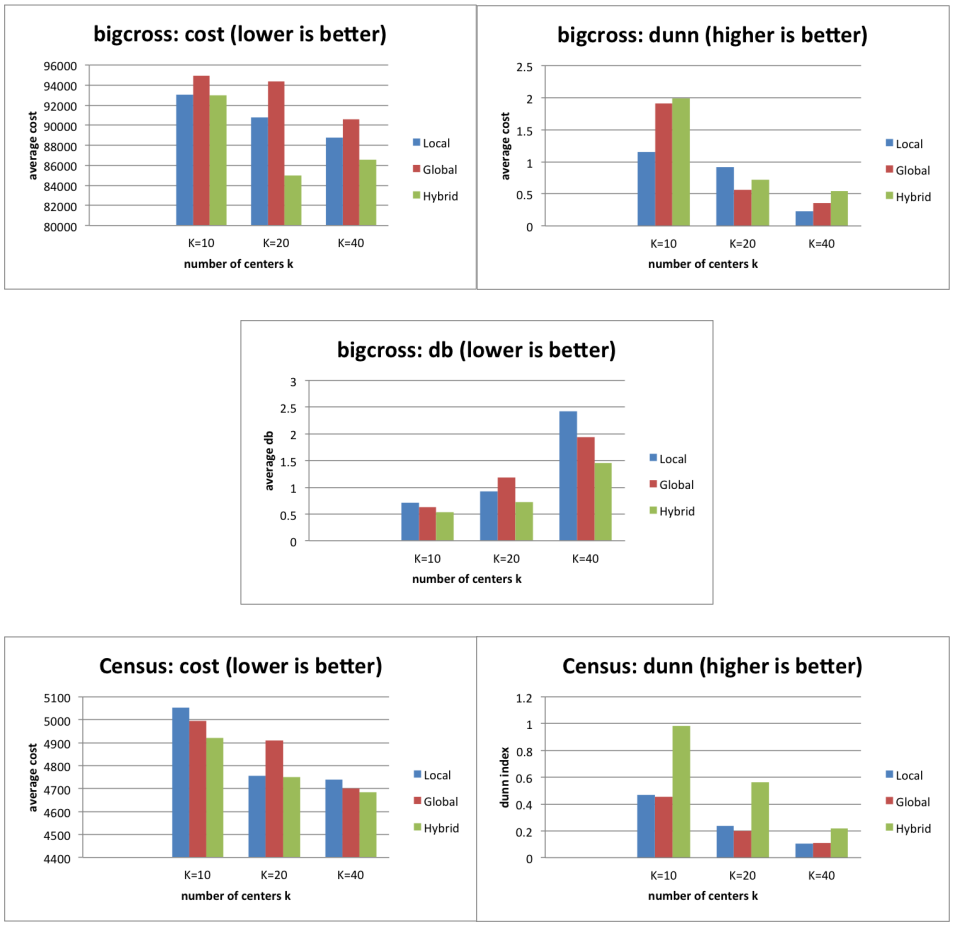
In these set of experiments, we measure the qualitative performance of our implementations on different datasets. To make a holistic evaluation, we use all three clustering quality metrics as mentioned in section 5.3.1.1.

An important point to note here is that, we have used 2 different types of convergence thresholds. This was done to show the effects of different convergence thresholds on the the results of clustering. To see the robustness in results, we have summarized the results for BigCross and Census on a lower convergence threshold in figure 6.2.

From figure 5.4, we observe that the hybrid performs better in almost all the cases. In some cases, the final quality of Global and Hybrid is lower than its local implementation, which is not ideal. This could be because both global and hybrid implementation were allowed to run only for 10 iterations that could be far from their convergence while local implementation ran till the end. The purpose of this experiment was to show that in most cases, hybrid implementation performs better, for a specified number of iterations.

So, it is fair to assume that as the quality is better in the hybrid for a fixed number of iterations, it is likely to converge faster than the batch. This is because KMeans works by improving the cost in every iteration and convergence usually happens upon plateauing of this cost metric.

We can also see that increasing k usually have lower values of Dunn Index while having Higher values of DB Index. A possible explanation could be that as the number of clusters increase, the well-separateness measured by Dunn decreases as well. Behavior of DB index with increasing k can also be explained similarly. This also depends on the distribution of the data and the number of actual clusters in the dataset.



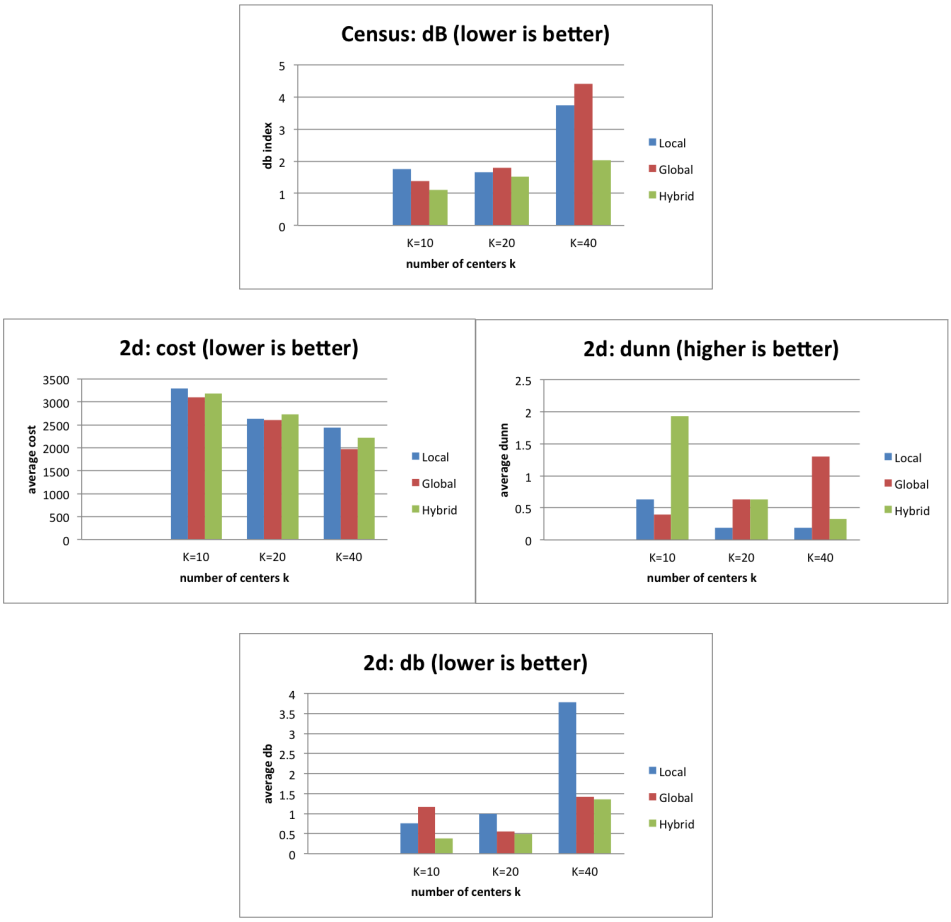


Figure 5.4: Qualitative Experimental Results

5.3.2 Classification

For Logistic Regression, along with the 6 general test programs, there is also a baseline test program that we choose: **LibLinear** [99]. Its a sequential classification library that outperforms most-other sequential implementations of Logistic Regression. We use it to train a sparse L1-regularized Logistic Regression model for the datasets in consideration. As before, due to the randomized nature of the online algorithms, the results of local and hybrid implementations are averaged over 3 runs. To check the correctness of the results, two runs were performed on the Batch implementations of the algorithms. The number of iterations for all types of experiments was fixed to 50. In order to maintain uniformity in results, similar parameters are taken as input for different training techniques for all

the 4 datasets. This is because, different parameters have different effects of optimization on different datasets. For example, figure 6.3 in the appendix shows the effect of choosing different learning rates. You can see that since the iterations are fixed, we had to rerun the experiments each time with a higher learning rate to show the applicability of the results.

5.3.2.1 Metrics

Fscore: Very often, the datasets are not balanced in terms of the number of labels from the positive class as compared to the negative class. For such unbalanced datasets, accuracy may not be a good criterion for evaluating the model.

Let's take the example[5] from our NER use case for identifying the "Person" class. Lets, say for 5000 sentences, after the feature extraction step, we came up with 1000 positive testing samples signifying that on an average a name occurs in every 5th sentence. Rest 100k samples were labeled as negative. After running, Logistic Regression classifier, we came up with the following confusion matrix-

	PREDICTED.PERSON	PREDICTED.NOT_PERSON
TRUE.PERSON	493	507
TRUE.NOT_PERSON	1017	98983

Figure 5.5: NER Example: Confusion Matrix

Where *P* refers to the *Person* class and *NP* refers to *Not-Person* class.

As we can see, from the 1000 positive samples, only 493 were labeled as being a Person, i.e. (True_Positive=493), the rest were labeled as Non-Persons (False_Positive=507). While, from 100k negative training examples, 98983 were labeled as Not_Persons and only 1017 were labeled as being Persons. As we can see, this model achieves an accuracy of 0.9849109. Now, how can this be possible that even after missing almost 50% of the labels of the Person class, the quality of the algorithm in terms of accuracy is 0.99. This

[5]example adapted from http://tata-box-blog.blogspot.de/

is called ***Accuracy paradox***. It usually occurs in a problem with a large class imbalance where a model can predict the value of the majority class for all predictions and achieve a high classification accuracy but finds no use in the target problem domain. For a detailed discussion on Accuracy paradox, readers are advised to visit [100].

So, accuracy doesn't solve our basic problem. Our basic problem here is to check whether we are detecting these comparatively fewer instances of Person class correctly. In order to do this, we have to come up with an evaluation metric which is much more focused on handling this imbalance correctly.

This is exactly what *Precision* and *Recall* do. Precision tells us that out of the things that we are selecting what percentage of selected items are correct; while recall tells us that out of the things that are correct, what percentage of them did we find [100].

$$Precision\ (exactness)\ = \frac{true\ positive}{true\ positive\ + false\ positive} \tag{5.6}$$

$$Recall\ (completeness)\ = \frac{true\ positive}{true\ positive + false\ negative} \tag{5.7}$$

As we can see, for our NER example, the precision comes out to be 0.3264901 while the recall is 0.493. Designers of such NLP applications, try to balance this trade-off by choosing one over the other based on the task at hand. But sometimes, using both these measures becomes very cumbersome in terms of both usage and analysis. ***F-Measure*** (F1-Measure / F1-Score/ F-Score) tries to remove this issue by combining the results of both these metrics into a single one by performing a weighted-average operation [100].

$$FScore = \frac{2 * Precision * Recall}{(Precision + Recall)} \tag{5.8}$$

As we can see, for our example, both the precision and recall values are not more than 0.50 each and our F-measure metric correctly combines it to 0.39. In general, the best F-score has its value at 1 and the worst score occurs at value 0.

The F-score is often the desired measure to be used in the field of information retrieval for measuring search, document classification, and query performance [34]. F-score has also been widely used in the field of natural language processing such as evaluation of named entity recognition and word segmentation [101]. The major advantage of F-score

lies in that it weights both precision and recall equally; hence, it favors good performance on both over extremely good performance on one and poor performance on the other. Below, we explain how we perform the computation of F-score for our experiments.

Sequential Validation. [102] provides additional packages called LibSVM tools that can be added to the native LibLinear library to add the functionality to measure F-Score. Readers are advised to go through [102] for detailed instructions for the same.

Local Validation. Once the local training is completed, the resultant local models are broadcasted to all the nodes, so that each node has all the models. Then testing is distributed over the nodes and for each such test instance a majority vote is computed by evaluating all the models for this instance following which the majority class is assigned. This majority vote is then used to measure the quality.

In general, for measuring quality using F1 metric, as discussed above, we need to keep track of 3 counters at the mapper. These are the number of *True positives* where the majority vote refers to the true value of positive, *False Positives* where majority vote is the positive class while the true value is the negative class and *False negative* which refers to the situation where the majority vote is negative class, but the true value is a positive class. The counts from all the test instances are then collected on a single reducer where they are aggregated to compute the final F Score. Figure 5.6 outlines the steps followed in the algorithm.

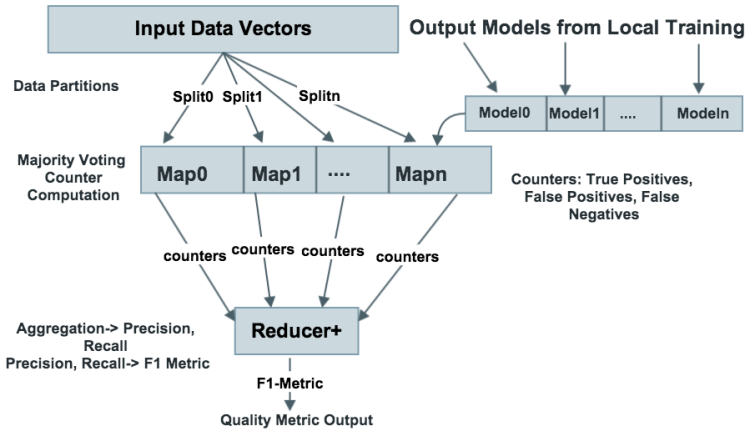


Figure 5.6: Local Validation Technique

Global/Hybrid Validation. The global and hybrid variants of validation techniques follow a similar workflow as the local one in terms of the computation happening at Mappers and Reducers. However, as we know, only a single model is outputted as the result of Global/ hybrid training; hence, there is no need for an extra step of computing the majority votes of the individual models to compute the respective counters. Figure 5.7 summarizes the overall validation algorithm for the global/ hybrid training.

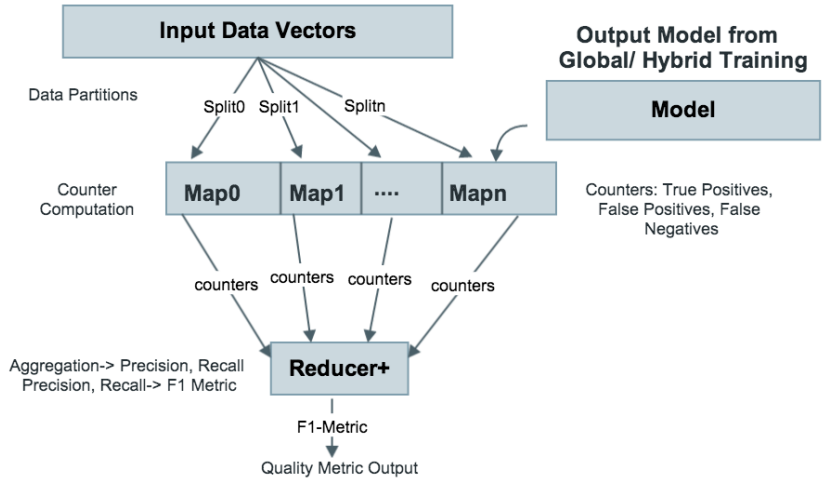


Figure 5.7: Global Validation Technique

5.3.2.2 Results on Running Time

Total Running Time	RCV1	20NewsGroup	Gisette	ClueWeb
Local	52	66	109	92
Global	1633	2691	2209	3736
Hybrid	1657	2777.5	2274	3859.5

Table 5.6: Comparison: Total Running Times

Our experiments on running time involved measuring the total running times for different implementations to run on different datasets. The purpose of these experiments

is to show the runtime performance of local when compared with the other training strategies. We can observe that as the size of the dataset increases the total running time of all our algorithms increase almost proportionally irrespective of the type of training involved. As expected, the *local* implementation is faster than the rest of the training strategies. *Batch* usually takes longer due to the inherent iterations, which are 50 in our case. Also, *hybrid* implementation takes a little extra time over the other iterations, which is expected for our case.

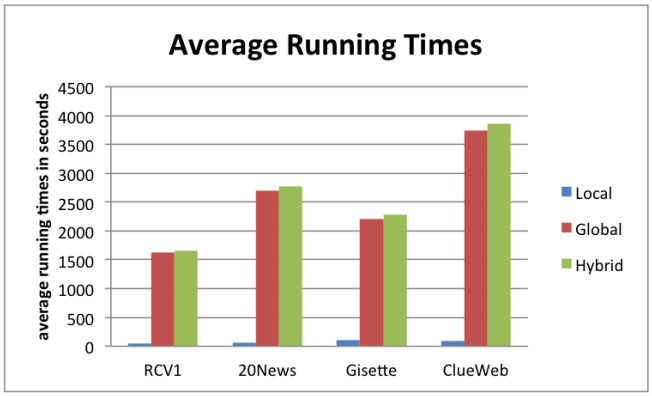


Figure 5.8: Average Running Times for 50 Iterations

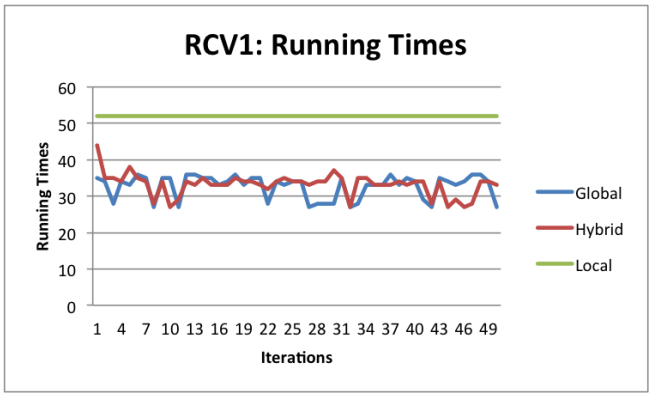


Figure 5.9: Time per Iteration: RCV1

Although, RCV1 is bigger than Gisette, but due to its sparseness it takes lesser time than Gisette. Hence, the running time not only depends on the number of dimensions but also on the inherent sparsity of the dataset. As mentioned in our runtime experiments on

clustering, a better way would be to check the total runtimes till convergence for all these approaches. As can be seen in figure 5.9 and figure 5.10, batch performs better in terms of runtime performance than local which is mainly due to inherent computation involved in learning multiple models in local against a single model in batch. Hybrid takes longer time in the first iteration due to the initial local step but then takes a similar time as the batch.

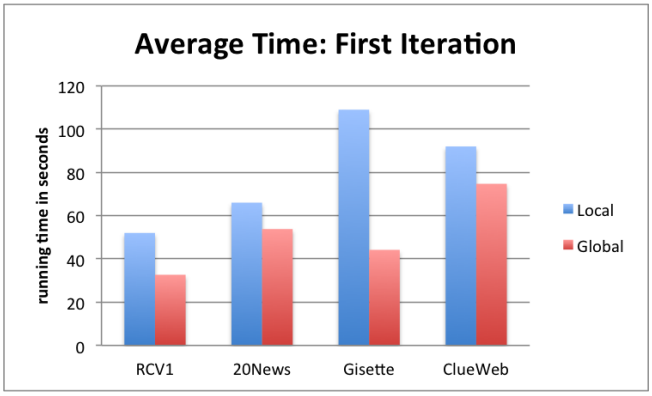


Figure 5.10: First iteration time: Comparison

5.3.2.3 Results on Precision

In this experiment, we show the results of our 3 test programs referring to 3 different training methodologies as discussed in chapter 3. Table 5.7 summarizes the average results of running the respective validation techniques on 5 datasets, described in section 5.2.1.

Fscore	RCV1	20NewsGroup	Gisette	ClueWeb
Local	0.948095814851017	0.698615173	0.881499022	0.965525989
Global	0.951465027	0.956251877	0.908366534	0.969288269
Hybrid	0.954906292	0.956358541	0.901375102	0.96895369
LibLinear	0.961412	0.974528	0.976884	0.57

Table 5.7: Comparison: Accuracy

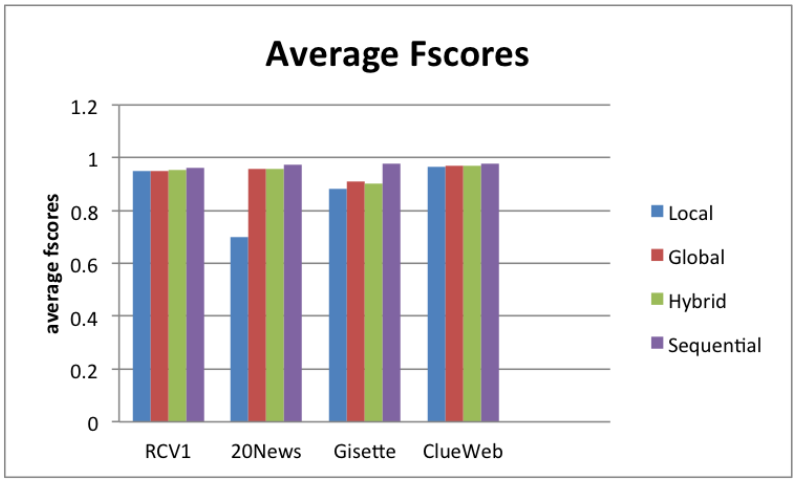


Figure 5.11: Average Total Fscores: Local, Global and Hybrid Training

As we observe from figure 5.11, the hybrid version performs slightly better than both local and batch algorithms for all the datasets. This is because of the warm-start or the initial push due to initial online pass. It can further be argued that both hybrid and batch implementations only perform slightly better than the local one. This could be due to the bounds involved in terms of iterations and how choice of learning rate can affect the delay in convergence. The purpose of this experiment was to show that for fixed number of iterations, hybrid can almost always perform better than global which can help in its faster convergence. This effect can be observed in figure 5.12, where for RCV1 dataset on a certain learning rate, hybrid approaches convergence faster due to the initial head-start provided by the local implementation. It can easily be seen that hybrid outperforms global by a difference of 32 iterations. As discussed initially, both hybrid and global trainings were performed using the same learning rate.

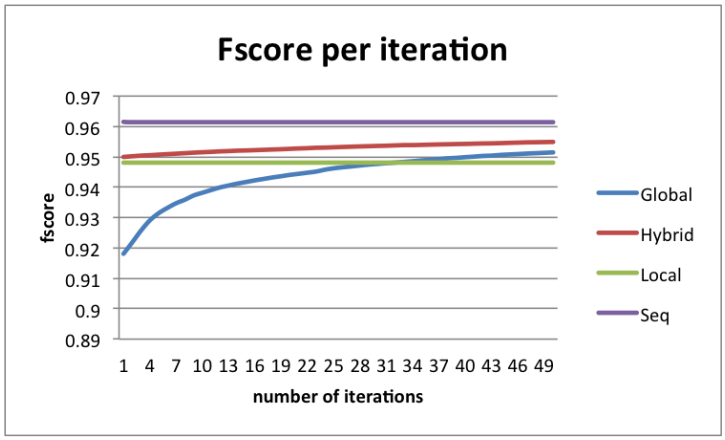


Figure 5.12: Fscore per Iteration: RCV1 dataset

5.3.2.4 Results on Ensemble Size

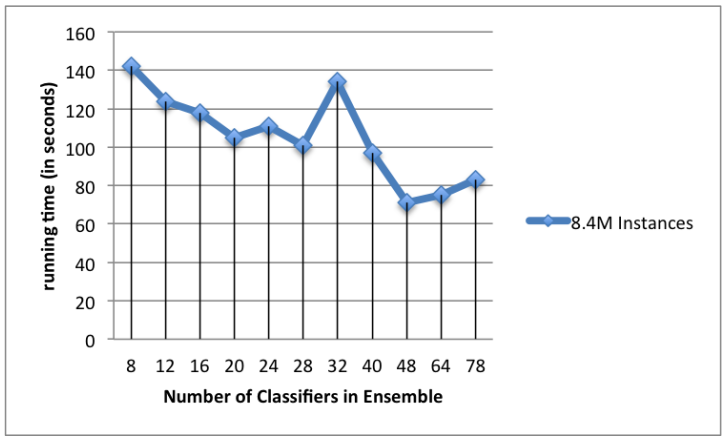


Figure 5.13: Effect of changing the ensemble size on the average running time

For the purpose of this experiment, we use a larger Kdda dataset available at [92]. Figure 5.13 shows the effect of increasing the degree of parallelism on the running time. As expected, the running time decreases as the degree of parallelism increases due to the decreasing "Straggler Effect." However, it becomes constant once the cost of setting up new nodes overcomes the performance gain achieved due to increased parallelism.

The main aim of this experiment is to validate the claim discussed in chapter 2 that an ensemble of classifiers trained on partitions of a large dataset outperforms a

single classifier trained on the entire dataset. This can be seen in figure 5.14, where an increase in the ensemble size increases the precision of the model. However, the benefits of an increase in ensemble size depends on the size of the dataset as well. For smaller datasets, a single classifier outperforms the ensemble of classifiers. It is evident from figure 5.15, where for smaller datasets such as Gisette increase in ensemble size proves to be counterproductive.

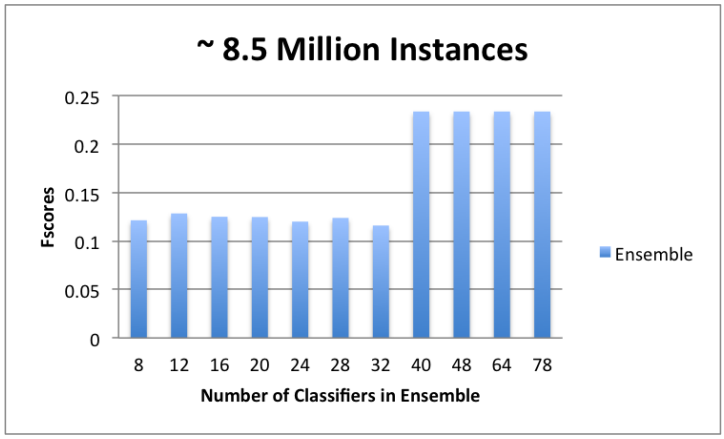


Figure 5.14: Effect of increasing the number of classifiers in Ensemble on Precision

This strategy is exploited by the machine learning department at Twitter [15]. As seen in section 2.3, they leverage the scaling ability of ensemble based training to boost the performance of their online training algorithms. However, still the qualitative performance of local training can't match the quality of models trained using batch training. In this case, using batch training, we can achieve a better Fscore of 0.39 in 30 iterations.

To compensate for this, we propose to use our hybrid approach that helps to achieve a better performance (0.43 in 23 iterations) due to the "warm-start" achieved using an initial online pass. This experiment motivates well the need for hybrid techniques for use-cases where emphasis is placed on both quality as well as overall performance. We would also like to point out that the reason for a low Fscore on Kdda is the poor choice of learning rate, which cases the algorithm to diverge. A possible solution could be to take a small random sample and select the best learning rate and then run it on the entire dataset.

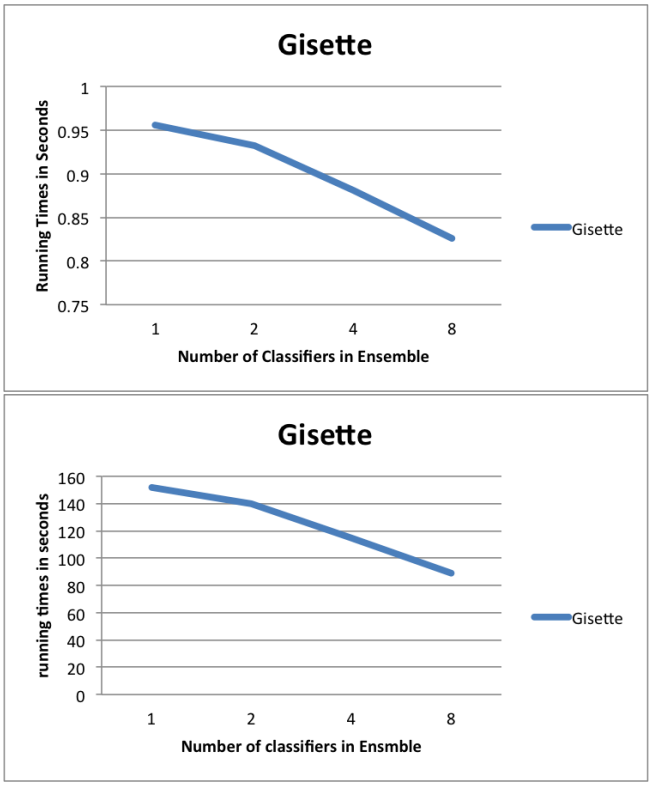


Figure 5.15: Effects of varying ensemble size on Gisette datasets

Chapter 6

Final Remarks

6.1 Conclusion

In this study, we presented an overview of the limitations of iterative learning algorithms on MapReduce. We discussed various works that targeted to resolve the limitations in performance by trading off speed with accuracy using approximate training techniques. However, this trade-off becomes unavoidable even in situation where there is a need for higher accuracy in return for a smaller compromise on speed.

The main idea of our approach lies around the claim that if MapReduce is not amenable to a particular class of algorithms then lets simply find a different class of algorithms that will solve the same problem and is amenable to MapReduce. However, we go one step further by proposing a "hybrid approach" that can serve as the "best of both the worlds." We propose that following our approach iterative Machine learning algorithms can achieve an accuracy that is comparable to the batch algorithms with a run time performance that is comparable to the online algorithms. The goal of this thesis is to analyze these three training approaches on two computing platforms on large-scale, high dimensional datasets for clustering and classification. We reached this goal with three major contributions-

First, we implemented all the three training approaches using both MapReduce and PACT programming paradigm. For this, we have implemented KMeans and Logistic Regression classifiers as the representative algorithms from both the Supervised and Unsupervised Learning fields of Machine Learning. The source code is available at:

https://github.com/GaurPrateek/ParallelML.

As our *second* contribution, we successfully presented a new approach that is builds upon the benefits of the initial approach and tries to remove its bottlenecks by leveraging the existing optimization techniques. Although none of the underlying techniques are new, but the careful design analysis required to obtain an efficient implementation is.

This leads to our *third* contribution, we evaluated our approach by conducting a series of experiments to test the proposed claim. The datasets for both clustering and classification were carefully chosen to represent many different use cases. For our empirical studies, we decided to stay within the confines of MapReduce paradigm. Based on extensive benchmarking, we summarized the effect of key features of each algorithm. Based on our empirical evaluation, hybrid training as a solution:

- achieves a better quality in relatively shorter span of time in comparison to global learning strategies
- is agnostic to the choice of scoring parameter
- achieves a better accuracy than the most optimized online training techniques
- proves counterproductive for smaller clustering datasets
- is sensitive to choice of hyperparameters
- scales well with the size of the dataset

6.2 Future Work

In the following, we briefly cover the potential areas for future work. We start with the work related to our Evaluation, continue with a possible future work for Implementation and finish with conceptual topics.

The hypothesis and the entire design methodology was developed during the thesis; hence, there was a limited time to check all the dimensions. We began working on our problem in March'14 when the most-stable release of Flink lacked the functionality of *Broadcast Variables*. The only workaround was to use a *Cross* operator that suffers from

the problem of quadratic complexity. Our implementation of the algorithms is available at [103]. Recent releases of Flink support the functionality of Broadcast variables. However, instead of focusing on optimizing our code, we focused more on doing a holistic set of experiments on Hadoop. At various points we also discussed, how various features of Flink are expected to behave and further compared the differences in programmability of Hadoop and Flink. Flink has proven to be an attractive alternative for iterative algorithms [33]. To test this, we propose to extend our implementations using the new framework (Broadcast Variables). It will be interesting to see how our proposed technique holds up on such a platform. In general, the trade-off of quality vs. performance should be agnostic to the choice of platforms. Hence, adding an in-memory platform such as *Spark* would provide a completeness to our study. Also, as discussed in section 2, different fault tolerance techniques are used by different platforms. For iterative algorithms, it will be interesting to see how our algorithmic-level approach affects the system level fault tolerance in different platforms.

Although, the choice of parameters was kept constant in all our experiments, there are still a lot of parameters that can lead to a sizable difference in performance of the underlying algorithms. An example could be Projection Search [43] that can lead to an even faster convergence of the online training techniques, but comes with a trade-off in terms of lower quality due to approximation. The results of our preliminary experiments suggest that the outputs are highly sensitive to the choice of such inputs. This could serve as a possible dealbreaker for a broader acceptance of our approach as all the local implementations are sensitive to a number of such hyperparameters. A possible extension could be to add an extra layer over the implementations of training techniques where the selection of trade-off can automatically set the required parameters. This can take the load off the designers who are more concerned with selecting the best algorithm/ platform for their use-case by evaluating these quality and performance trade-offs.

Although, we used a 9-node cluster but our experimental learnings are expected to hold up for larger datasets and more computing nodes as well. A possible way to extend our experimental evaluation would be to gauge its validity in a production setting with larger computing nodes to deal with the larger datasets. This will also help in benchmarking our approach of parallel algorithm-based training against the architecture-

level training performed in [65]. In our experiments, we had seen how hybrid can produce better quality results faster than the global approach. It will be interesting to evaluate our hypothesis empirically that achieving better quality faster leads to an overall faster *convergence*.

At the implementation level, we propose to optimize our existing implementations by learning from the issues we faced during the course of the study. Some such improvements could be to add an extra MapReduce job for a proper random partitioning in Stochastic Gradient Descent. This can be extended to perform sampling with replacement to incorporate true ensemble learning. Adding support for *L2-regularization* in our implementations for Logistic Regression and trying to see the effect of *partitioning based on feature sets* for ensemble-based learning are few other implementation-level optimizations that can be performed. The Hybrid training approach can further be extended by weighting the classifiers based on the confidence of their predictions before carrying out the global step.

The area of potential research in large-scale predictive analytics is quite wide. We discussed how more and more platforms are extending their platforms to embrace machine-learning solutions. We further learned the benefits of letting the application drive development of new features by trying to solve the problems using whatever we have before trying to develop entire new solutions. In this study, we discussed how supervised and unsupervised learning problems benefit from the proposed framework. However, many companies such as Google are also applying predictive analytics on semi-supervised learning techniques such as Spam detection. We expect our treatment on classification and clustering problems to hold up on other popular areas of machine learning such as semi-supervised learning and ranking.

Appendix A

Java implementations of the algorithms described in this report are available at [103]. The package includes a considerable amount of software for generating test distributions for KMeans and dataset parsing packages for ClueWeb dataset for Logistic Regression. The key classes of Interest are explained in the table below:

Class of Interest	**Package**	**MainClass**
MapReduce Local Logistic Regression	mapredLocalLogreg	EnsembleJob
MapReduce Global Logistic Regression	mapredGloballLogreg	BatchGradientDriver
MapReduce Hybrid Logistic Regression	mapredHybridLogreg	EnsembleBatch
PACT Local Logistic Regression	pactLocalLogreg	EnsembleJob
PACT Global Logistic Regression	pactGlobalLogreg	BatchGradientDriver
PACT Local Logistic Regression	pactHybridLogreg	EnsembleBatch
MapReduce Local KMeans	mapredLocalKMeans	StreamingRun
MapReduce Global KMeans	mapredGlobalKMeans	BatchRun
MapReduce Hybrid KMeans	mapredHybridKMeans	StreamingBatch
PACT Local KMeans	pactLocalKMeans	OnePassPlan
PACT Global KMeans	pactGlobalKMeans	KMeansIterative
PACT Hybrid KMeans	pactHybridKMeans	OnePassIterative
Simulated Clustering Data Generator	datagenerator	LibSVMDataGenerator
NER Feature Extractor	dima.ner	DataReader
Testing Scripts Location	testUtils	TestingScripts

Table 6.1: Classes of Interest Description

Appendix B

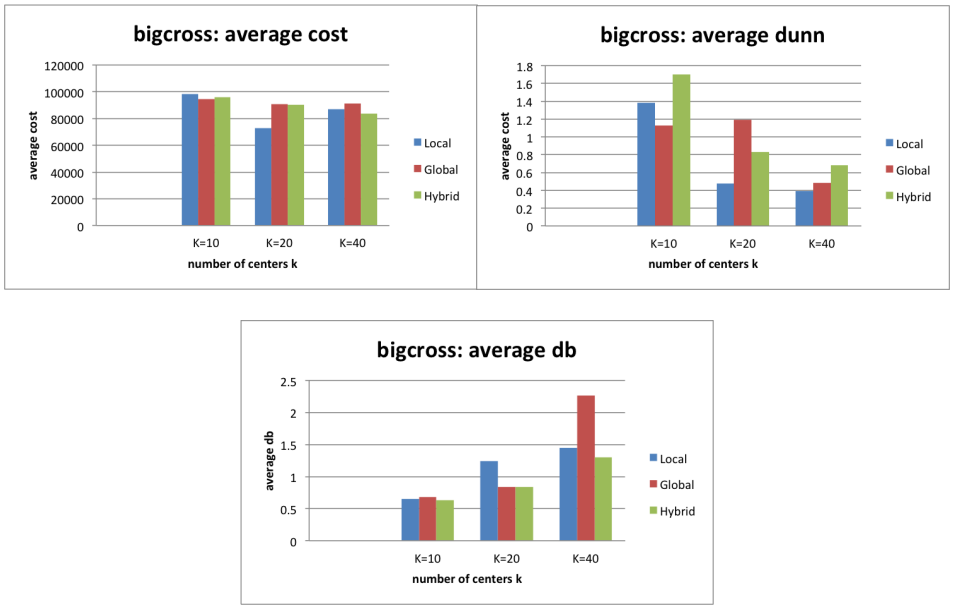
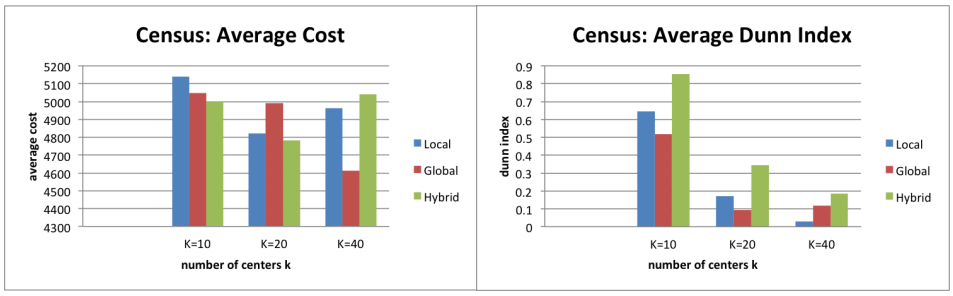


Figure 6.1: BigCross: Cluster-Quality Results on a convergence threshold of 0.01



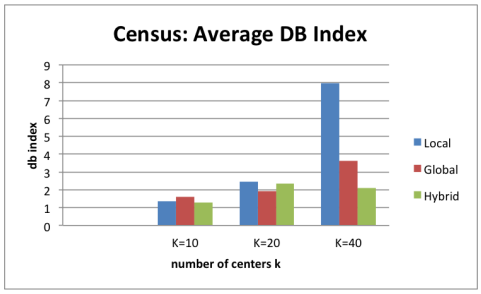


Figure 6.2: Census: Cluster-Quality Results on a convergence threshold of 0.01

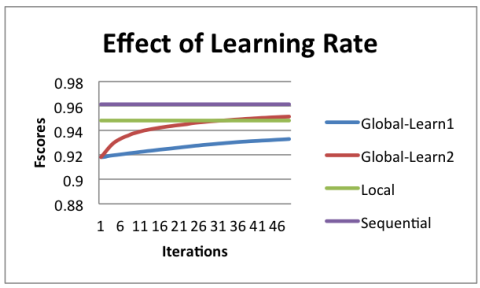


Figure 6.3: Effect of different learning rates on Accuracy

Bibliography

[1] W. Redmond, "http://www.microsoft.com/en-us/news/features/2013/feb13/02-11bigdata.aspx," August 2014.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] T. White, *Hadoop: the definitive guide: the definitive guide.* " O'Reilly Media, Inc.", 2009.

[4] D. Filimon, "Clustering big data," 2013.

[5] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in Neural Information Processing Systems*, 2010, pp. 2595–2603.

[6] C. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," *Advances in neural information processing systems*, vol. 19, p. 281, 2007.

[7] M. Kearns, "Efficient noise-tolerant learning from statistical queries," *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 983–1006, 1998.

[8] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman, "Vision paper: Towards an understanding of the limits of map-reduce computation," *arXiv preprint arXiv:1204.1754*, 2012.

[9] A. D. Sarma, F. N. Afrati, S. Salihoglu, and J. D. Ullman, "Upper and lower bounds on the cost of a map-reduce computation," in *Proceedings of the VLDB Endowment*, vol. 6, no. 4. VLDB Endowment, 2013, pp. 277–288.

[10] M. Stonebraker, "The case for shared nothing," *IEEE Database Eng. Bull.*, vol. 9, no. 1, pp. 4–9, 1986.

[11] D. Borthakur, "Hdfs architecture guide," *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

[12] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 439–455.

[13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 10–10.

[14] J. Lin, "Mapreduce is good enough? if all you have is a hammer, throw away everything that's not a nail!" *Big Data*, vol. 1, no. 1, pp. 28–37, 2013.

[15] J. Lin and A. Kolcz, "Large-scale machine learning at twitter," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 793–804.

[16] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl *et al.*, "The stratosphere platform for big data analytics," *The VLDB Journal*, pp. 1–26, 2014.

[17] A. Alexandrov, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp, and D. Warneke, "Mapreduce and pact-comparing data parallel programming models." in *BTW*, 2011, pp. 25–44.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[19] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel, "Amazon s3 for science grids: a viable solution?" in *Proceedings of the 2008 international workshop on Data-aware distributed computing*. ACM, 2008, pp. 55–64.

[20] S. Owen, R. Anil, T. Dunning, and E. Friedman, *Mahout in action*. Manning, 2011.

[21] H. T. Docs, "http://hadoop.apache.org/docs/r1.2.1/streaming.html," July 2014.

[22] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of map-reduce: the pig experience," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1414–1425, 2009.

[23] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive-a petabyte scale data warehouse using hadoop," in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE, 2010, pp. 996–1005.

[24] M. Islam, A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur, "Oozie: towards a scalable workflow management system for hadoop," in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*. ACM, 2012, p. 4.

[25] Blog-entry, "http://bimentalist.com/," July 2014.

[26] "http://hortonworks.com/hadoop-tutorial/hello-world-an-introduction-to-hadoop-hcatalog-hive-and-pig/," July 2014.

[27] A. Hacker, "Evaluating parallel data processing systems for scalable feature selection," Master's thesis, ï¿ŒFU Berlin, 2013.

[28] M. Leich, J. Adamek, M. Schubotz, A. Heise, A. Rheinländer, and V. Markl, "Applying stratosphere for big data analytics." in *BTW*, 2013, pp. 507–510.

[29] F. Hueske, M. Peters, A. Krettek, M. Ringwald, K. Tzoumas, V. Markl, and J. Freytag, "Peeking into the optimization of data flow programs with mapreduce-style udfs," in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE, 2013, pp. 1292–1295.

[30] D. Warneke and O. Kao, "Nephele: Efficient parallel data processing in the cloud," in *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, ser. MTAGS '09. New York, NY, USA: ACM, 2009, pp. 8:1–8:10.

[31] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/pacts: a programming model and execution framework for web-scale analytical processing," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 119–130.

[32] R. Metzger, "http://robertmetzger.de/stratosphere/docs/0.4/," March 2014.

[33] S. Ewen, S. Schelter, K. Tzoumas, D. Warneke, and V. Markl, "Iterative parallel data processing with stratosphere: an inside look," in *Proceedings of the 2013 international conference on Management of data*. ACM, 2013, pp. 1053–1056.

[34] S. M. Beitzel, "On understanding and classifying web queries," Ph.D. dissertation, Illinois Institute of Technology, 2006.

[35] W. Ben Abdessalem Karaa, "Named entity recognition using web document corpus," *arXiv preprint arXiv:1102.5728*, 2011.

[36] D. M. Allen, "Mean square error of prediction as a criterion for selecting variables," *Technometrics*, vol. 13, no. 3, pp. 469–475, 1971.

[37] T. G. Dietterich, "Ensemble methods in machine learning," in *Multiple classifier systems*. Springer, 2000, pp. 1–15.

[38] I. Jolliffe, *Principal component analysis*. Wiley Online Library, 2005.

[39] R. Ostrovsky, Y. Rabani, L. J. Schulman, and C. Swamy, "The effectiveness of lloyd-type methods for the k-means problem," *Journal of the ACM (JACM)*, vol. 59, no. 6, p. 28, 2012.

[40] M. R. Ackermann, M. Märtens, C. Raupach, K. Swierkot, C. Lammersen, and C. Sohler, "Streamkm++: A clustering algorithm for data streams," *Journal of Experimental Algorithmics (JEA)*, vol. 17, pp. 2–4, 2012.

[41] T. Dunning, "Kmeans clustering at scale," March 2014.

[42] A. Blum, "Random projection, margins, kernels, and feature-selection," in *Subspace, Latent Structure and Feature Selection*. Springer, 2006, pp. 52–68.

[43] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. ACM, 1998, pp. 604–613.

[44] S.-I. Lee, H. Lee, P. Abbeel, and A. Y. Ng, "Efficient l˜ 1 regularized logistic regression," in *Proceedings of the National Conference on Artificial Intelligence*, vol. 21, no. 1. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006, p. 401.

[45] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman, *Applied linear statistical models*. Irwin Chicago, 1996, vol. 4.

[46] L. R. Example, "http://www.simafore.com/," April 2014.

[47] J. C. Platt, "Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods," in *Advances in large margin classifiers*. Citeseer, 1999.

[48] D. W. Hosmer, S. Lemeshow, and R. X. Sturdivant, *Introduction to the logistic regression model*. Wiley Online Library, 2000.

[49] H. Bruck, S. McNeill, M. A. Sutton, and W. Peters Iii, "Digital image correlation using newton-raphson method of partial differential correction," *Experimental Mechanics*, vol. 29, no. 3, pp. 261–267, 1989.

[50] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal, "Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization," *ACM Transactions on Mathematical Software (TOMS)*, vol. 23, no. 4, pp. 550–560, 1997.

[51] A. Pfeffer, "Cs181 lecture 5: Perceptrons," 2014.

[52] M. A. Hearst, S. Dumais, E. Osman, J. Platt, and B. Scholkopf, "Support vector machines," *Intelligent Systems and their Applications, IEEE*, vol. 13, no. 4, pp. 18–28, 1998.

[53] S. Haykin and N. Network, "A comprehensive foundation," *Neural Networks*, vol. 2, no. 2004, 2004.

[54] J. C. Dunn, "A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters," 1973.

[55] P. Melville and R. J. Mooney, "Creating diversity in ensembles using artificial data," *Information Fusion*, vol. 6, no. 1, pp. 99–111, 2005.

[56] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[57] Y. Freund, R. E. Schapire *et al.*, "Experiments with a new boosting algorithm," in *ICML*, vol. 96, 1996, pp. 148–156.

[58] J. Lin and C. Dyer, "Data-intensive text processing with mapreduce," *Synthesis Lectures on Human Language Technologies*, vol. 3, no. 1, pp. 1–177, 2010.

[59] A. Halevy, P. Norvig, and F. Pereira, "The unreasonable effectiveness of data," *Intelligent Systems, IEEE*, vol. 24, no. 2, pp. 8–12, 2009.

[60] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, and S. Vishwanathan, "Hash kernels for structured data," *The Journal of Machine Learning Research*, vol. 10, pp. 2615–2637, 2009.

[61] J. Langford, A. Smola, and M. Zinkevich, "Slow learners are fast," *arXiv preprint arXiv:0911.0491*, 2009.

[62] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems*, 2011, pp. 693–701.

[63] R. Mcdonald, M. Mohri, N. Silberman, D. Walker, and G. S. Mann, "Efficient large-scale distributed training of conditional maximum entropy models," in *Advances in Neural Information Processing Systems*, 2009, pp. 1231–1239.

[64] R. McDonald, K. Hall, and G. Mann, "Distributed training strategies for the structured perceptron," in *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, 2010, pp. 456–464.

[65] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford, "A reliable effective terascale linear learning system," *arXiv preprint arXiv:1110.4198*, 2011.

[66] A. Agarwal and J. C. Duchi, "Distributed delayed stochastic optimization," in *Advances in Neural Information Processing Systems*, 2011, pp. 873–881.

[67] M. Banko and E. Brill, "Scaling to very very large corpora for natural language disambiguation," in *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 2001, pp. 26–33.

[68] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean, "Large language models in machine translation," in *In Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. Citeseer, 2007.

[69] C. Dyer, A. Cordova, A. Mont, and J. Lin, "Fast, easy, and cheap: Construction of statistical machine translation models with mapreduce," in *Proceedings of the Third Workshop on Statistical Machine Translation*. Association for Computational Linguistics, 2008, pp. 199–207.

[70] M. Dredze, A. Kulesza, and K. Crammer, "Multi-domain learning by confidence-weighted parameter combination," *Machine Learning*, vol. 79, no. 1-2, pp. 123–149, 2010.

[71] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 2010, pp. 810–818.

[72] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[73] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[74] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 2011, pp. 1151–1162.

[75] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[76] S. G. Nash and A. Sofer, "Block truncated-newton methods for parallel optimization," *Mathematical Programming*, vol. 45, no. 1-3, pp. 529–546, 1989.

[77] H. Peng, D. Liang, and C. Choi, "Evaluating parallel logistic regression models," in *Big Data, 2013 IEEE International Conference on*. IEEE, 2013, pp. 119–126.

[78] L. D. I. Team, "Data infrastructure at linkedin," in *Proc. of ICDE*, 2012.

[79] D. Sculley, M. E. Otey, M. Pohl, B. Spitznagel, J. Hainsworth, and Y. Zhou, "Detecting adversarial advertisements in the wild," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2011, pp. 274–282.

[80] M. D. Intriligator, *Mathematical optimization and economic theory*. Siam, 2002, vol. 39.

[81] E.-G. Talbi and T. Muntean, "Hill-climbing, simulated annealing and genetic algorithms: a comparative study and application to the mapping problem," in *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, vol. 2. IEEE, 1993, pp. 565–573.

[82] J. M. Borwein and A. S. Lewis, *Convex analysis and nonlinear optimization: theory and examples*. Springer, 2010, vol. 3.

[83] K. B. Hall, S. Gilpin, and G. Mann, "Mapreduce/bigtable for distributed optimization," in *NIPS LCCC Workshop*, 2010.

[84] E. Amazon, "Amazon elastic mapreduce," 2012.

[85] P. S. Efraimidis, "Weighted random sampling over data streams," *arXiv preprint arXiv:1012.0256*, 2010.

[86] Stratosphere, "https://stratosphere2.dima.tu-berlin.de/wiki/doku.php/wiki:kmeansexample," July 2014.

[87] M. Shindler, A. Wong, and A. W. Meyerson, "Fast and accurate k-means for large datasets," in *Advances in neural information processing systems*, 2011, pp. 2375–2383.

[88] T. G. Dietterich, "Machine-learning research," *AI magazine*, vol. 18, no. 4, p. 97, 1997.

[89] S. C. Bagui, "Combining pattern classifiers: methods and algorithms," *Technometrics*, vol. 47, no. 4, pp. 517–518, 2005.

[90] T. Joachims, "Svmlight: Support vector machine," *SVM-Light Support Vector Machine http://svmlight. joachims. org/, University of Dortmund*, vol. 19, no. 4, 1999.

[91] S. Ewen, "http://flink.incubator.apache.org/," 2014.

[92] A. Asuncion and D. Newman, "Uci machine learning repository. university of california, school of information and computer science, irvine, ca (2007)."

[93] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li, "Rcv1: A new benchmark collection for text categorization research," *The Journal of Machine Learning Research*, vol. 5, pp. 361–397, 2004.

[94] H.-F. Yu, H.-Y. Lo, H.-P. Hsieh, J.-K. Lou, T. G. McKenzie, J.-W. Chou, P.-H. Chung, C.-H. Ho, C.-F. Chang, Y.-H. Wei *et al.*, "Feature engineering and classifier ensemble for kdd cup 2010," 2010.

[95] J. Callan, M. Hoy, C. Yoo, and L. Zhao, "Clueweb09 data set," 2009.

[96] E. Gabrilovich, M. Ringgaard, and A. Subramanya, "Facc1: Freebase annotation of clueweb corpora, version 1 (release date 2013-06-26, format version 1, correction level 0)," 2013.

[97] D. Bär, T. Zesch, and I. Gurevych, "Dkpro similarity: An open source framework for text similarity." in *ACL (Conference System Demonstrations)*. Citeseer, 2013, pp. 121–126.

[98] D. L. Davies and D. W. Bouldin, "A cluster separation measure," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, no. 2, pp. 224–227, 1979.

[99] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "Liblinear: A library for large linear classification," *The Journal of Machine Learning Research*, vol. 9, pp. 1871–1874, 2008.

[100] X. Zhu and I. Davidson, *Knowledge Discovery and Data Mining: Challenges and Realities*. Igi Global, 2007.

[101] D. M. Powers, "Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation," 2011.

[102] C.-J. Lin, "http://www.csie.ntu.edu.tw/ cjlin/libsvmtools/," April 2014.

[103] P. Gaur, "https://github.com/gaurprateek/parallelml," accessed July'14.

List of Abbreviations

DAG Directed acyclic graph

HDFS Hadoop Distributed File System

PACT Parallelization Contract

JVM Java Virtual Machine

UDF User Defined Function

LR Logisted Regression

SGD Stochastic Gradient Descent

List of Tables

List of Figures

Listings