Table of Contents

1. Introduction 1.1
2. Operations and Transformations 1.2
    1. Calculating Average 1.2.1
    2. Calculating Average per Key 1.2.2
    3. Application: Word Count 1.2.3
    4. Joins 1.2.4
    5. Application: New Articles 1.2.5
    6. Application: PageRank 1.2.6
3. Basic Statistics 1.3
4. Classification 1.4
5. Support Vector Machines 1.5
6. Logisitic Regression 1.6
7. Linear Regression 1.7
    1. Statistical Simple Linear Regression 1.7.1
    2. Optimisation 1.7.2
    3. Gradient Descent 1.7.3
    4. Implementation 1.7.4
    5. Spark Implementation 1.7.5
8. Multilayer Perceptron 1.8

Introduction

Hands-On Machine Learning and Big Data

In Scala, Java, Python and C++

Version 1

Kareem Alkaseer

MentorLycon

[Cover image: Flickr user guillermo varela]

Operations and Transformations

Operations and Transformations

Spark Setup

Spark can be run in cluster mode using either Mesos or Yarn and it also has its own basic cluster manager which also allows it to run locally.

The easiest way to try Spark out is to download Spark binary named Spark without Hadoop. Make the necessary con guration, start master and worker, then you are ready to go. Spark does not need Hadoop to run but it needs HDFS (or a compatible lesystem). Spark spills data to the lesystem and it must be distributed and reliable, HDFS serves this purpose. Also this allows Spark to load and save data to HDFS.

Here are some basic con gurations to get things up, all of them reside in the conf subdirectory of the Spark binary directory

custom-env.sh

```
#!/usr/bin/env bash
export SPARK_HOME=PATH TO SPARK DIRECTORY ROOT
export PYSPARK_PYTHON=python
export HADOOP_HOME=PATH TO HADOOP DIRECTORY ROOT
export PATH=.:$PATH:$SPARK_HOME/bin:$HADOOP_HOME/bin
export SPARK_DIST_CLASSPATH=$(hadoop classpath)
export JAVA_LIBRARY_PATH=$JAVA_LIBRARY_PATH:$HADOOP_HOME/lib/native:$
export SPARK_LIBRARY_PATH=$SPARK_LIBRARY_PATH:$SPARK_HOME/lib
export LD_LIBRARY_PATH=$LD_LIBARARY_PATH:PATH TO SNAPPY LIBARARY BIN/
. ~/.bash_profile
. conf/spark env.sh
```

spark-env.sh

```
SPARK_LOCAL_IP=127.0.0.1
SPARK_LOCAL_DIRS=$SPARK_HOME/work
SPARK_MASTER_IP=127.0.0.1
STANDALONE_SPARK_MASTER_HOST=127.0.0.1
SPARK_MASTER_HOST=127.0.0.1
SPARK_WORKER_CORES=2
SPARK_WORKER_INSTANCES=2
SPARK_WORKER_MEMORY=2g
SPARK_WORKER_DIR=$SPARK_HOME/work
SPARK_EXECUTOR_INSTANCES=1
```

spark-default.conf

```
spark.master
spark.serializer
spark.driver.memory
```

```
spark.driver.extraJavaOptions XX:+PrintGCDetails Dkey=value Dnumbers=
Dorg.xerial.snappy.tempdir=/tmp/snappy
spark.driver.extraClassPath $SPARK_HOME/lib/:$HADOOP_HOME/share/hadoo
spark.driver.extraLibraryPath PATH TO SNAPPY LIBARARY BINARIES:$SPARI
spark.executor.extraClassPath $SPARK_HOME/lib/:$HADOOP_HOME/share/hac
```

log4j.properties

```
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss}
log4j.logger.org.spark project.jetty=WARN
log4j.logger.org.spark project.jetty.util.component.AbstractLifeCycle
log4j.logger.parquet=ERROR
log4j.logger.org.apache.hadoop.hive.metastore.RetryingHMSHandler=FATA
log4j.logger.org.apache.hadoop.hive.ql.exec.FunctionRegistry=ERROR
```

To start Spark

```
$ HADOOP_HOME/sbin/start dfs.sh
$ . SPARK_HOME/conf/custom env.sh
$ SPARK_HOME/sbin/start master.sh
$ SPARK_HOME/sbin/start slaves.sh
```

To start Spark shell and py-spark

```
$ SPARK_HOME/bin/pyspark
```

Spark Transformations and Actions

Transformations are descriptions of how to generate a new RDD from input RDDs, i.e. they are operation descriptors. Transformations are lazy evaluated whenever an action is executed. Actions actually execute all the pending transformations and then apply their own operations.

Transformations

`map()`

Applies a function to each element of the input RDD. The function is not restricted in its return value type but it must return a single value. For example a function may take as input a string value and return word count, i.e. input is RDD[String], output is RDD[Integer], or take a URL as input and fetch it, i.e. input is RDD[String] output RDD[Array[Byte]].

```
val input = sc.parallelize(List(1,2,3,4))
val squared = input.map(x => x * x)
```

```
input = sc.parallelize([1, 2, 3, 4])
squared = input.map(lambda x: x * x)
```

```
JavaRDD<Integer> input = sc.parallelize(Arrays.asList(1, 2, 3, 4))
JavaRDD<Integer> squared = input.map(new Function<Integer, Integer>(
public Integer call(Integer x) { return x * x; } });
```

`flatMap()`

Applies a function to each element of the input RDD similarly to map() but returns an iterator instead of a single value. The output RDD is composed of all the values in all iterators. For example, atMap() can be used to split words in each line of a string and returns an RDD composed of all the individual words.

For a list of strings [Spark is cool , distributed computing], the output RDD would be composed of [Spark , is , cool , distributed , computing]

```
val lines = sc.parallelize(List( "Spark is cool",  "distributed compu
val words = lines.flatMap(line => line.split(""))

lines = sc.parallelize([ "Spark is cool",  "distributed computing" ]
words = lines.flatMap(lambda line: line.split(""))

JavaRDD<String> lines = sc.parallelize(Arrays.asList( "Spark is cool'
JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, St
    public Iterable<String> call(String line) {
        return Arrays.asList(line.split(""));
    }
});
```

Pseudo set operations

Some set transformations will include duplicate elements if the input RDDs contain duplicates, other will remove duplicates. The `distinct()`, `intersection()` and `subtract()` transformations eliminate duplicates. Removing duplicates is expensive because it requires shuffling RDD elements across the cluster to guarantee uniqueness. Set transformations that operate on more than one set require the input RDDs to be of the same type. `cartesian()` is very expensive for large RDDs.

```
// RDD1: [ dog, dog, cat, horse, dolphin ]
// RDD2: [ dog, horse, whale ]

RDD1.union(RDD2) // [ dog, dog, cat, horse, dolphin, dog, horse, wha

RDD1.intersection(RDD2) // [ dog, horse ]

RDD1.subtract(RDD2) // [ dolphin ]

RDD1.distinct() // [ dog, cat, horse, dolphin ]

RDD1.cartesian(RDD2)
// [
//    (dog, dog), (dog, horse), (dog, whale),
//    (dog, dog), (dog, horse), (dog, whale),
//    (cat, dog), (cat, horse), (cat, whale),
//    (horse, dog), (horse, horse), (horse, whale),
//    (dolphin, dog), (dolphin, horse), (dolphin, whale),
// ]
```

`filter()`

Returns an RDD of elements satisfying certain criteria.

```
val filtered = rdd.filter(x => x != 1)
```

```
filtered = rdd.filter(lambda x: x != 1)
```

```
JavaRDD<Integer> filtered = rdd.filter(new Function<Integer, Integer>
    public Integer call(Integer x) { return x != 1; }
});
```

`sample(withReplacement, fraction, [seed])`

Samples an RDD with or without replacement.

`sortByKey()`

Sorts data by their key according to their natural ordering if no comparator is provided or according to the provided comparator. The most general overload is `sortByKey(comparator, ascending, numPartitions)`.

```
val input: RDD[(Int, String)] = ...
implicit val sortIntegerAsString = new Ordering[Int] {
override def compare(a: Int, b: Int) = a.toString.compare(b.toString
input.sortByKey(sortIntegerAsString)
```

```
input.sortByKey(ascending=True, numPartitions=None, keyfunc = lambda
```

```
JavaPairRDD<Integer, String> input = ...
class IntegerAsStringComparator implements Comparator<Integer> {
    public int compare(Integer a, Integer b) {
        return String.valueOf(a).compareTo(String.valueOf(b));
    }
}
input.sortByKey(new IntegerAsStringComparator());
```

Actions

`reduce()`

Takes two elements of the RDD and return a single value of the same type.

```
val sum = rdd.reduce((x, y) => x + y)
```

```
sum = rdd.reduce(lambda x, y: x + y)
```

```
Integer sum = rdd.reduce(new Function2<Integer, Integer, Integer>()
    public Integer call(Integer x, Integer y) { return x + y; }
});
```

`fold()`

Similar to `reduce()`, takes two elements of the RDD and an extra zero value of the same type as the

RDD type to return a single value of the same type. The zero type should be the identity value for the operation to be carried out. For example, 0 for addition, for 1 for multiplication, empty list for concatenation.

Object allocation could be minimised in `fold()` by modifying the first parameter in place and returning it. The second parameter should not be modified. The function would act differently from non-distributed fold operations if the function is not commutative because the fold may be applied locally on each partition then results would be fold into the final result.

```
val sum = rdd.fold(0, (x, y) => x + y)
val mul = rdd.fold(1, (x, y) => x * y)
```

```
sum = rdd.fold(0, lambda x, y: x + y)
mul = rdd.fold(1, lambda x, y: x * y)
```

```
Integer sum = rdd.fold(
    Integer zero, new Function2<Integer, Integer, Integer>() {
        public Integer call(Integer x, Integer y) { return x + y; }
});
```

Calculating Average

Calculating Average

Using map() and reduce()

```
val pairs = rdd.map(x => (x, 1))
val result = pairs.reduce((t1, t2) => (t1._1 + t2._1, t1._2 + t2._2)
val average = result._1 / result._2.toDouble
```

```
pairs = rdd.map(lambda x: (x, 1))
result  = pairs.reduce(lambda t1, t2: (t1[0] + t2[0], t1[1] + t2.[1]
average = result[0] / float(result[1])
```

```
JavaPairRDD<Integer, Integer> pairs = rdd.mapToPair(
    new PairFunction<Integer, Integer, Integer>() {
        public Tuple2<Integer, Integer> call(Integer x) { return new
    });
Tuple2<Integer, Integer> result = pairs.reduce(
    new Function2<
        Tuple2<Integer, Integer>,
        Tuple2<Integer, Integer>,
        Tuple2<Integer, Integer>
    > () {
        public Tuple2<Integer, Integer> call(
            Tuple2<Integer, Integer> t1, Tuple2<Integer, Integer> t2
        ) {
            return new Tuple2<>(t1._t1 + t2._t2, t1_.t2 + t2._t2);
        }
    });
double average = result._t1 / (double) result._t2;
```

Using aggregate()

```
val result = rdd.aggregate((0, 0))(
    (acc, value) => (acc._1 + value, acc._2 + 1),
    (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
val average = result._1 / result._2.toDouble
```

```
result = rdd.aggregate(
    (0, 0),
    lambda acc, value: (acc[0] + value, acc[1] + 1),
    lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1]))
average = result[0] / float(result[1])
```

```
class Average implements Serializable {
    public Average(int x, int count) {
        this.x = x;
        this.count = count;
```

```
    }
    public double average() { return x / (double) count; } public int
    public int count;
}

Function2<Average, Integer, Average> addAndCount = new Function2<>()
    public Average call(Average average, Integer x) {
        average.x += x;
        average.count += 1;
        return average;
    }
};

Function2<Average, Average, Average> combine = new Function2<>() {
    public Average call(Average a1, Average a2) {
        a1.x += a2.x;
        a1.count += a2.count;
        return a1;
    }
};

Average result = rdd.aggregate(new Average(0, 0), addAndCount, combi
```

Calculating Average per Key

Calculating Average per Key

Using mapValues() and reduceByKey()

```
rdd.mapValues(x => (x, 1)).reduceByKey((x, y) => (x._1 + y._1, x._2 +

rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x, y: (x[0] + y[0]

// class Average as before
JavaPairRDD<Integer, Average> pairs = pairsRdd.mapValues( new Functi
    public Tuple2<Integer, Average> call(Integer x) { return new Ave
};

JavaPairRDD<Integer, Average> perKey = pairs.reduceByKey( new Functi
    public Average call(Average a, Average b) {
        a.x += b.x;
        a.count += b.count;
        return a;
    }
};
```

Using combineByKey()

```
val result = pairRdd.combineByKey(
    // createCombiner: key first seen in partition
    (v) => (v, 1),
    // mergeValue: key is known, add to accumulator for the same key
    (acc: (Int, Int), v) => (acc._1 + v, acc._2 + 1),
    // mergeCombiners: merge combiners from different partitions for
    (acc1: (Int, Int), acc2: (Int, Int) => (acc1._1 + acc2._2, acc1._
result.map((key, value) => (key, value._1 / value._2.toDouble))

result = pairRdd.combineByKey(
    (lambda x: (x, 1)),
    (lambda x, v: (x[0] + v, x[1] + 1)),
    (lambda x, y: (x[0] + y[0], x[1] + y [1])))
result.map(lambda key, value: (key, value[0] / float(value[1])))

// class Average as before
Function<Integer, Average> createCombiner = new Function<>() {
    public Average call(Integer x) { return new Average(x, 1); }
};

Function2<Average, Integer, Average> mergeValue = new Function2<>() {
    public Average call(Average a, Integer x) {
        a.x += x;
        a.count += 1;
```

```
        return a;
    }
};

Function2<Average, Average, Average> mergeCombiners = new Function2<
    public Average call(Average a, Average b) {
        a.x += b.x;
        a.count += b.count;
        return a;
    }
};

JavaPairRDD<String, Average> averages = pairRdd.combineByKey( create(
```

Application: Word Count

Application: Word Count

Method 1

```
sc.textFile(...)
    .flatMap(x => x.split(""))
    .map(x => (x, 1))
    .reduceByKey((x, y) => x + y)
```

```
sc.textFile(...)
    .flatMap(lambda x: x.split(""))
    .map(lambda x: (x, 1))
    .reduceByKey(lambda x, y: x + y)
```

```
JavaPairRDD<String, Integer> result = sc.testFile(...)
    .flatMap(new FlatMapFunction<String, String>() {
        public Iterable<String> call(String x) { return Arrays.asLis
    })
    .mapToPair(new PairFunction<String, String, Integer>() {
        public Tuple2<String, Integer> call(String x) { return new Tu
    })
    .reduceByKey(new Function2<Integer, Integer, Integer>() {
        public Integer call(Integer a, Integer c) { return a + b; }
    });
```

Method 2

```
sc.textFile(...).flatMap(x => x.split(   )).countByValue()
```

```
sc.textFile(...).flatMap(lambda x: (x, 1)).countByValue()
```

```
sc.textFile(...)
    .flatMap(new FlatMapFunction<String, String>() {
        public Iterable<String> call(String x) { return Arrays.asLis
    }).mapToPair(new PairFunction<String, String, Integer>() {
        public Tuple2<String, Integer> call(String x) { return new Tu
    }).countByValue();
```

Joins

Joins

Spark offers multiple types of joins out of the box. Given the following two collections:

```
  c1 = [ (technology, Peter), (machine learning, Anna), (science,
  Alex) ] c2 = [ (technology,4), (machine learning,2),
  (science,3), (neural networks,1) ]
```

The following types of joins are covered by Spark API.

```
c1.join(c2) = [
    (technology, (Peter,4)), (machine learning, (Anna,2)), (science,

c1.leftOuterJoin(c2) = [
     (technology, (Peter, Some(4))), (machine learning, (Anna, Some(2
     (science, Alex, Some(3))) ]

c1.rightOuterJoin(c2) = [
     (technology, (Some(Peter),4),
     (machine learning, (Some(Anna),2),
     (science, (Some(Alex),3),
     (neural networks, (None,1))
]
```

Application: New Articles

Application: New Articles

Assume there is a Hadoop sequence file containing pairs `<UserId, UserInfo>`. `UserInfo` contains a list of articles a user has read before. During a new session the user reads other articles. The session is short-lived on average. Articles read during session are represented by `<UserId, Article>`. We need to know the new articles the user read during the session. The computation is carried out periodically.

```
val sc =newSparkContext(...)
val data = sc.sequenceFile( hdfs://...")              (1)
 .partitionBy(newHashPartitioner(100))                (2)
 .persist()                                           (3)

def getNewArticles() {
   val articles: RDD[(UserId, Article)] = ...
   val joined = data.join(articles)                    (4)
   val newArticles = joined.filter {
   case(userId, (userInfo, article)) =>
       !userInfo.topics.contains(article)
   }
   val perUser = newArticles.reduceByKey(x => x._2)
}
```

The implementation reduces the number of shuffles needed as follows

1. User data is contained in a HadoopSequenceFile.
2. The computation is carried out periodically and the user data is supposedly large, we avoid shuffling it each time when we call `getNewArticles()` by assigning a hash partitioner to the RDD so that Spark knows where the data is and partitions the dependent RDDs accordingly.
3. `persist()` ensures that we use the pre-partitioned RDD.
4. `join()`, `filter()` and `reduceByKey()` will use the partitioner on the data RDD.

Operations Break-down

```
data = [
 (user1, {..., [ article1, article2 ]} ),
 (user2, {..., [ article1, article3 ]} ),
 (user3, {..., [ article4, article5 ]} )
]

articles = [
 (user1, article1),
 (user1, article6),
 (user2, article1),
 (user3, article6),
```

```
 (user3, article7)
]

joined = [
 (user1, ( {..., [ article1, article2 ]}, article1 )),
 (user1, ( {..., [ article1, article2 ]}, article6 )),
 (user2, ( {..., [ article1, article3 ]}, article1 )),
 (user3, ( {..., [ article4, article5 ]}, article6 )),
 (user3, ( {..., [ article4, article5 ]}, article7 ))
]

newArticles = [
 (user1, ( {..., [ article1, article2 ]}, article6)),
 (user3, ( {..., [ article4, article5 ]}, article6)),
 (user3, ( {..., [ article4, article5 ]}, article7))
]

perUser = [ (user1, article6), (user3, article6), (user3, article7) ]
```

Application: PageRank

Application: PageRank

The idea of PageRank is to calculate the relative importance of a node in a graph based on in-links from other nodes. The node may be a web page, a profile in a social network or a document in a linked documents database, etc.

The algorithm given here is a little di erent from strict one. Let's first begin with a standalone faithful implementation of PageRank. Using sparse matrices would be more e cient but for clarity let's do plain Java.

```
public class PageRankStandalone {

    private double alpha = 0.85;
    private double epsilon = 0.001;
    double[] pageRankVector;
    HMatrix h;
    int size; double inverse;

    public PageRankStandalone(int nPages) {
        size = nPages;
        h = new HMatrix(nPages);
    }

    public double pageRank(String url) {
        int i = h.indices().index(url);
        return i < pageRankVector.length && i >= 0 ? pageRankVector[
    }

    public HMatrix matrix() { return h; }

    public void init() {
        inverse = size > 0 ? 1.0d / size : 1.0d / 0.000001d;
        calculate(alpha, epsilon);
    }

    public void calculate(double alpha, double epsilon) {
        int i = 0;
        int j = 0;
        double[][] G = h.matrix();
        double error = 1.0;
        pageRankVector = new double[size];
        double[] previousVector = new double[size];

        for (i = 0; i < size; ++i) pageRankVector[i] = 1.0d; double[
        double randomJump = (1 - alpha) * inverse;

        // transform H into G
        for (i = 0; i < size; ++i)
```

```
            for (j = 0; j < size; ++j)
                G[i][j] = alpha * G[i][j] + dangling[i][j] + randomJu

        while (error >= epsilon) {
            System.arraycopy(
                pageRankVector, 0, previousVector, 0, pageRankVector
            for (i = 0; i < size; ++i) {
                acc = 0.0;
                for (j = 0; j < size; ++j)
                    acc += previousVector[j] * G[j][i];
                    pageRankVector[i] = acc;
            }
            error = norm(pageRankVector, previousVector);
        }
    }

    private double norm(double[] a, double[] b) {
        double norm = 0.0;
        for (int i = 0; i < a.length; ++i)
            norm += Math.abs(a[i] + b[i]);
        return norm;
    }

    private double[][] danglingMatrix() {
        double inverse = 1.0d / size;
        boolean[] d = h.dangling();
        double[][] nodes = new double[size][size]; for (int i = 0; i
        for (int j = 0; j < size; ++i)
            nodes[i][j] = d[i] ? alpha * inverse : 0;
      return nodes;
    }

    public static class ValueIndexMapping {

        private int nextIndex = 0;
        private Map<String, Integer> values = new HashMap<>();
        private Map<Integer, String> indices = new HashMap<>();

        public ValueIndexMapping() { }

        public int index(String value) {
            Integer index = values.get(value);
            if (index == null) {
                index = nextIndex; ++nextIndex; values.put(value, ind
            }
            return index;
        }

        public String value(int index) {
            return indices.get(index);
        }

        public int size() { return values.size(); }
```

```
    }

    public static class HMatrix {

        private double[][] matrix;
        private int nDanglingPages = 0;
        private ValueIndexMapping valueIndexMapping = new ValueIndexM

        public HMatrix(int nPages) { matrix = new double[nPages][nPag

        public void addLink(String url) { valueIndexMapping.index(url

        public void addLink(String fromUrl, String toUrl, double weig
            int i = valueIndexMapping.index(fromUrl);
            int j = valueIndexMapping.index(toUrl);
            try {
                matrix[i][j] = weight;
            } catch (final ArrayIndexOutOfBoundsException e) { }
        }

        void calculate() {
            for (int i = 0; i < matrix.length; ++i) {
                double rowSum = 0.0;
                for (int j = 0; j < matrix.length; ++j)
                    rowSum += matrix[i][j];
                if (rowSum > 0) {
                    for (int j = 0; j < matrix.length; ++j)
                    matrix[i][j] = matrix[i][j] / rowSum;
                } else {
                    ++nDanglingPages;
                }
            }
        }

        public boolean[] dangling() {
            int n = size();
            boolean[] a = new boolean[n];
            boolean ok;
            for (int i = 0; i < n; ++i) {
                ok = false;
                for (int j = 0; j < n; ++j) {
                    if (matrix[i][j] > 0) {
                        ok = true;
                        break;
                    }
                }
                a[i] = ok;
            }
            return a;
        }

    public int size() { return matrix.length; }
```

```
    public double[][] matrix() { return matrix; }

    public ValueIndexMapping indices() { return valueIndexMapping; }
}
```

Spark Naive Implementation

```
class DomainPartitioner(count: Int) extends Partitioner {

  override def numPartitions: Int = count

  override def getPartition(key: Any): Int = {
    val domain = new java.net.URL(key.toString).getHost()
    val hcode = domain.hashCode % numPartitions
    if (hcode < 0) { hcode + numPartitions } else { hcode }
  }

  override def equals(other: Any): Boolean = other match {
    case dp: DomainPartitioner =>
      dp.numPartitions == numPartitions
    case _ =>
      false
  }
}

object PageRank {

  def main(args: Array[String]) {

    val conf = new SparkConf().setMaster("local").setAppName("PageRa
    val sc = new SparkContext(conf)

    val links = sc.parallelize(List(
      ("http://url1", List("http://url2", "http://url3")),
      ("http://url2", List("http://url3", "http://url4")),
      ("http://url5", List("http://url6", "http://url7"))
    )).partitionBy(new DomainPartitioner(100))
      .persist()

    var ranks = links.mapValues(l => 1.0)

    for (i <- 0 until 10) {
      val contribs = links.join(ranks).flatMap {
        case (src, (links, rank)) =>
          links.map(dest => (dest, rank / links.size))
      }

      val mapped = contribs.reduceByKey((x,y) => x+y).mapValues(v =>
      ranks = ranks.union(mapped).reduceByKey((x, y) => x + y)
      val total = ranks.values.reduce(_+_)
      ranks = ranks.mapValues(x => x / total)
    }
```

```
  println("ranks: " + ranks.collect().mkString( ,"))
  ranks.saveAsTextFile( ranks ) }
}
```

Spark Efficient Implementation

Spark GraphX includes two implementations of PageRank: rst runs for a speci ed number of iterations and the second runs until the algorithm converges. Refer to https://spark.apache.org/docs/1.1.0/graphx-programming-guide.html for more details. Here is an implementation copied verbatim from the page referenced above.

```
val graph = GraphLoader.edgeListFile(sc,  graphx/data/followers.txt
val ranks = graph.pageRank(0.0001).vertices
val users = sc.textFile( graphx/data/users.txt ).map {
      line =>
      val fields = line.split(",")
      (fields(0).toLong, fields(1))
}

val rankByUsername = users.join(ranks).map {
      case (Id, (username, rank)) => (username, rank)
}

println(ranksByUsername.collect.mkString( \n ))

// And here is an efficient implementation of the PageRank algorithm

def efficient() {
      val conf = new SparkConf().setMaster("local").setAppName("Pagel
      val sc = new SparkContext(conf)
      val links: RDD[(VertexId, String)] = sc.parallelize( Array(
            (1L, "http://url1"),
            (2L, "http://url2"),
            (3L, "http://url3"),
            (4L, "http://url4"),
            (5L, "http://url5"),
            (6L, "http://url6"),
            (7L,  http://url7")))
      val edges: RDD[Edge[Int]] = sc.parallelize( Array(
            Edge(1L, 2L, 0),
            Edge(1L, 3L, 0),
            Edge(2L, 3L, 0),
            Edge(2L, 4L, 0),
            Edge(5L, 6L, 0),
            Edge(5L, 7L, 0)))
      val defaultUrl = "http://none"
      val graph = Graph(links, edges, defaultUrl)
      val ranks = graph.pageRank(0.0001).vertices
      println(ranks.collect.mkString("\n") }
```

Basic Statistics

Basic Statistics

Let's code it ourselves first to get a feel of how it looks like.

```
#include <functional>
#include <algorithm>
#include <numeric>
#include <cmath>
#include <vector>

template< class T >
struct to_double {
    using real_type = T;
    double operator()(const T & a) { return static_cast<double>(a); }
};

template<
    class RAIterator,
    class Comparator = std::greater<typename RAIterator::value_type>,
    class Plus       = std::plus<typename RAIterator::value_type>,
    class Minus      = std::minus<typename RAIterator::value_type>,
    class Multiply   = std::multiplies<typename RAIterator::value_ty
    class Divide     = std::divides<typename RAIterator::value_type>,
    class Real       = to_double<typename RAIterator::value_type> >

class statistics {

public:

    using iterator    = RAIterator;
    using comparator  = Comparator;
    using minus       = Minus;
    using plus        = Plus;
    using multiply    = Multiply;
    using divide      = Divide;
    using real        = Real;
    using value_type  = typename iterator::value_type;

    statistics() { }

    statistics(iterator first, iterator last)
        : m_first{ first },
          m_last{ last }
    {}

    value_type median() {

        if (m_median_value.ok)
```

```
            return m_median_value.value;

        size_t length = m_last - m_first;

        if ((length & 1) == 0) {
            std::nth_element(m_first, m_first + length / 2, m_last);
            m_median_upper = optional_iterator{ m_first + length / 2
            m_median_lower = optional_iterator{ m_median_upper.value
            m_median_value = optional_value{
                divide{}(plus{}(*m_median_upper.value, *m_median_lowe
        } else {
            std::nth_element(m_first, m_first + length / 2, m_last);
            m_median_upper = m_median_lower = optional_iterator{ m_fi
            m_median_value = optional_value{ *m_median_upper.value } ;
        }

        return m_median_value.value;
    }

    iterator nth(std::size_t n) {

        std::nth_element(m_first, m_first + n, m_last);
        return m_first + n;
    }

    iterator q1() {

        if (m_q1.ok)
            return m_q1.value;

        if (! m_median_value.ok) median();

        std::nth_element(
            m_first,
            m_first + (m_median_lower.value - m_first) / 2,
            m_median_lower.value); m_q1 = optional_iterator{
                m_first + (m_median_lower.value - m_first) / 2 };

        return m_q1;
    }

    iterator q3() {

        if (m_q3.ok)
            return m_q3.value;

        if (! m_median_value.ok) median();

        std::nth_element(
            m_median_upper.value,
            m_median_upper.value + (m_last - m_median_upper.value) /
            m_last);
```

```
        m_q3 = optional_iterator{
            m_median_upper.value + (m_last - m_median_upper.value) /

        return m_q3;
    }

    size_t iqr() { return minus{}(*q3(), *q1()); }

    template< class Vector = std::vector<value_type> >
    Vector minor_outliers() {

        auto iqrv = iqr();
        auto lower = *q1() - (iqrv * 1.5);
        auto upper = *q3() + (iqrv * 1.5);
        std::vector<value_type> result; std::copy_if(
            m_first, m_last, std::back_inserter(result),
            [lower, upper](const value_type & a) { return a < lower

        return result;
    }

    template< class Vector = std::vector<value_type> >
    Vector major_outliers() {

        auto iqrv = iqr();
        auto lower = *q1() - (iqrv * 3.0);
        auto upper = *q3() + (iqrv * 3.0);
        std::vector<value_type> result; std::copy_if(
            m_first, m_last, std::back_inserter(result),
            [lower, upper](const value_type & a) { return a < lower

        return result;
    }

    value_type mean() {

        if (m_mean_value.ok)
            return m_mean_value.value;

        m_mean_value = optional_value{
            divide{}(std::accumulate(m_first, m_last, 0, plus{}), (m_

        return m_mean_value.value;
    }

    template< class Vector = std::vector<value_type> >
    value_type stddev() {

        if (m_std_dev.ok)
            return m_std_dev.value;

        real r{};
        auto add = plus{};
```

```
        auto subtract = minus{};
        auto times = multiply{};
        auto m = mean();
        auto length = m_last - m_first; Vector v(length);

        std::transform(
            m_first,
            m_last,
            v.begin(),
            [&r, &subtract, &m](const value_type & x) { return r(sub

        value_type product = std::inner_product(
            v.begin(), v.end(), v.begin(), 0, add, times);

        // -1 for sample standard deviation
        m_std_dev = optional_value{ std::sqrt(product / (length - 1)

        return m_std_dev.value;
    }

    value_type stddev_insensitive() {

        if (m_std_dev.ok)
            return m_std_dev.value;

        real r{};
        auto add = plus{};
        auto subtract = minus{};
        auto m = mean();
        typename real::real_type s = 0;
        std::for_each(
            m_first,
            m_last,
            [&](const value_type & x) { s = add(s, std::pow(r(subtra

        // -1 for sample standard deviation
        m_std_dev = optional_value{ std::sqrt(1.0 / (m_last - m_firs

        return m_std_dev.value;
    }

private:

    template< class T >
    struct optional {
        explicit optional() : ok{ false }, value{} { }
        explicit optional(T && v) : ok { true }, value{ std::forward
        explicit optional(T & v) : ok { true }, value{ v } { }
        operator T() { return value; }
        bool ok;
        T value;
    };
```

```
    using optional_value = optional<value_type>;
    using optional_iterator = optional<iterator>;

    iterator          m_first;
    iterator          m_last;
    ptional_iterator  m_median_lower;
    optional_iterator m_median_upper;
    optional_iterator m_q1;
    optional_iterator m_q3;
    optional_value    m_median_value;
    optional_value    m_mean_value;
    optional_value    m_std_dev;
};
```

That's good summary statistics, but the outlier tests are basic. Both tests would not detect an outlier for the sequence `{ 5, 6, 4, 3, 2, 6, 7, 9, 3 }`. Let's see how to do a chi-square test in R.

// TODO: implement chi-square in C++

```
> library(outliers)
> v <- c(5,6,4,3,2,6,7,9,3)
> chisq.out.test(v)

    chi-squared test for outlier
    data: v
    X-squared = 3.2, p-value = 0.07364
    alternative hypothesis: highest value 9 is an outlier

> chisq.out.test(v, opposite=T)

    chi-squared test for outlier
    data: v
    X-squared = 1.8, p-value = 0.1797
    alternative hypothesis: lowest value 2 is an outlier
```

The good news is that our previous outliers tests agree with the chi-square test, with a p-value = 0.07364 and a signi cance level of 0.05, we do not have enough evidence to reject the null hypthesis of no outliers. And with a p-value = 0.1797, we do not have enough evidence to recognise the lowest value 2 as an outlier.

Classification

Classification

Let's look at a problem and try to solve it using a linear classifier that we devise based on intuition.

Assume we have a data set of pairs of people with their characteristics, interests and whether they match. For the sake of discussion let's have age, drinker, interests, location, and match.

Age is numeric, drinker is {-1, 0, 1} where 0 means missing, each interest is textual, location is a<latitude, longitude>pair, match is a {0, 1}.

The linear classi er's job is to nd a straight-line separating matching pairs from non-matching pairs. A simple classi er may nd the average point of all instances in a class and predicts unseen data based on the average distance from the class's cluster's centre.

We have to turn textual data to numerical values somehow. Three natural ways to think of this case are: 1) a vector containing all interests in order, an element is 1 if they share the interest or 0 if they don't, 2) the collective number of shared interests, or 3) a hierarchal tree of interests type. For simplicity, let's choose the second.

```
#define _USE_MATH_DEFINES
#include <cmath>
#include <string>
#include <vector>
#include <istream>
#include <sstream>
#include <unordered_map>
#include <algorithm>
#include <iterator>
#include <numeric>
#include <limits>

/* Earth radius in Km */
#define EARTH_RADIUS 6371

using namespace std;

static std::vector<std::string> split(const std::string & s, char de

    vector<string> out;
    string token;
    istringstream ss(s);
    while (getline(ss, token, delim))
        out.push_back(token);
    return out;
}

static double to_radians(double degrees) { return degrees *(M_PI / 1
```

```
class row {
public:

    explicit row(vector<string> features1, vector<string> features2,
        : match{ matches }
    {
        data.push_back(stod(features1[0])); data.push_back(stod(featu
        data.push_back(stod(features1[0]) * stod(features1[0]));
        data.push_back(location_proximity(features1[2], features2[2]
        data.push_back(interests(
            features1.begin() + 3, features1.end(),
            features2.begin() + 3, features2.end()));
    }

    explicit row(size_t size, double x)
        : data(size, x), match{ false }
    {}

    row() : match{ false } { }

    static vector<row> load(istream s) {

        vector<row> rows;
        string line1, line2;
        while (true) {
        getline(s, line1);
        if (s.good()) {
                getline(s, line2);
                if (s.good()) {
                    std::vector<string> row2 = split(line2, ',');
                    bool match = stoi(row2.back());
                    row2.pop_back();
                    rows.push_back(row(split(line1, ','), row2, matcl
                } else {
                    break;
                }
            }
            else break;
        }

        return rows;
    }

    // Haversine distance
    double location_proximity(string s1, string s2) {

        vector<string> v1 = split(string{ s1.begin() + 1, s1.end() -
        vector<string> v2 = split(string{ s2.begin() + 1, s2.end() -
        double latitude1  = stod(v1[0]);
        double longitude1 = stod(v1[1]);
        double latitude2  = stod(v2[0]);
        double longitude2 = stod(v2[1]);
        double lat_diff  = to_radians(latitude1 - latitude2);
```

```
        double lng_diff  = to_radians(longitude1 - longitude2);
        double a = sin(lat_diff / 2) * sin (lat_diff / 2) +
            cos(to_radians(latitude1)) * cos(to_radians(latitude2)) 
            sin(lng_diff / 2) * sin(lng_diff / 2);
        double c = 2 * atan2(sqrt(a), sqrt(1-a));

        return EARTH_RADIUS * c;
    }

    double static interests(
        vector<string>::iterator begin1,
        vector<string>::iterator end1,
        vector<string>::iterator begin2,
        vector<string>::iterator end2
    ) {
        sort(begin1, end1);
        sort(begin2, end2);
        vector<string> intersection;
        set_intersection(begin1, end1, begin2, end2, back_inserter(i

        return static_cast<double>(intersection.size());
    }

    double dot(const row & other) {
        return inner_product(data.begin(), data.end(), other.data.be
    }

    size_t size() { return data.size(); }

    double & operator[](size_t n) { return data[n]; }

    vector<double>::iterator begin() { return data.begin(); }

    vector<double>::iterator end() { return data.end(); }

    vector<double> data;
    bool match;
};

class dataset {
public:

    explicit dataset(vector<row> rows)
        : data{ rows },
          low{ data[0].size(), numeric_limits<double>::max() },
          high{ data[0].size(), numeric_limits<double>::min() }
    {
        // find global minimum and maximum values for each feature t
        for (auto & r : data) {
            for (size_t i = 0; i < r.size(); ++i) {
                if (r[i] < low[i]) low[i] = r[i];
                if (r[i] > high[i]) high[i] = r[i];
            }
```

```
        }

        for (auto & r : data)
            scale(r);
    }

    // scale each feature in each row to be between [0,1]
    void scale(row & r) {
        for (size_t i = 0; i < r.size(); ++i)
            r[i] = (r[i] - low[i]) / (high[i] - low[i]);
    }

    size_t row_size() { return data[0].size(); }

    vector<row> data; row low;
    row high;
};

class linear_model {
public:

    explicit linear_model() { }

    void train(dataset & ds) {

        averages.clear();
        averages.emplace(true, row{ ds.row_size(), 0.0 });
        averages.emplace(false, row{ ds.row_size(), 0.0 });
        int match_count = 0;
        int nonmatch_count = 0;

        for (auto & row : ds.data) {
            for (size_t i = 0; i < ds.row_size(); ++i)
                averages[row.match][i] += row[i];

            if (row.match)
                ++match_count;
            else
                ++nonmatch_count;
        }

        for (auto it = averages.begin(); it != averages.end(); ++it)
            int count = it->first ? match_count : nonmatch_count;
            for (auto & feature : it->second)
                feature /= count;
        }
    }

    bool classify(row point) {
        double b =
            (averages[false].dot(averages[false]) - averages[true].d
        double y = point.dot(averages[false]) - point.dot(averages[t
        return y > 0;
```

```
    }

    unordered_map<bool, row> averages;
};
```

Support Vector Machines

Support Vector Machines

SVM are a collection of rich and efficient classification methods. One of their excellent attributes is ease of reasoning about how they work.

The basic idea of SVM is that two-fold:

1. Having a linearly inseparable dataset, we may well be able to map the dataset from the current n-dimensional space to a new m-dimensional space in which the dataset is linearly separable. For example, the XOR function is linearly inseparable in 2 dimensions, however, if we map it from $\{x_1, x_2\}$ to the 6-dimensional feature space $\{1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2, x_1^2, x_2^2\}$ it is linearly separable. Linear separability is achieved by hyperplanes which are a generalisation of a strightlines.

2. Having found a separating hyperplane, how do we test the its tness? In other words, how to nd the optimal separating hyperplane? It turns out that the best hyperplane has the greatest distance from all closest data points from all di erent classes. That is, it has the largest margin from support vectors.

In Spark version 1.6, only linear SVM are supported and only binary classi ers are available out-of-the-box.

The objective function is of the form

$$f(w) = \lambda R(w) + \frac{1}{n}\sum_{i=1}^{n} L(w, x_i, y_i)$$

Loss function is of the form

$$f(w, x, y) = max\{0, 1 - y\ w^T x\}$$

where w is the d-dimensional weights vector or the learnt parameters vector, x is the d-dimensional input vector and y is the output class.

$R(w)$ is the regularisation term $\ell 2$ defined by default as

$$\frac{1}{n}||w||_2^2$$

```
import org.apache.spark.mllib.classification.{SVMModel, SVMWithSGD}
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.mllib.util.MLUtils

// Load training data in LIBSVM format.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data
```

```
// Split data into training (60%) and test (40%).
val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0).cache()
val test = splits(1)

// Run training algorithm to build the model
val numIterations = 100
val model = SVMWithSGD.train(training, numIterations)
// Clear the default threshold.
model.clearThreshold()

// Compute raw scores on the test set.
val scoreAndLabels = test.map { point =>
    val score = model.predict(point.features)
    (score, point.label)
}

// Get evaluation metrics.
val metrics = new BinaryClassificationMetrics(scoreAndLabels) val auI
println("Area under ROC = '' + auROC)

// Save and load model
model.save(sc, "myModelPath")
val sameModel = SVMModel.load(sc, "myModelPath")
```

Logisitic Regression

Logisitic Regression

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.classification.LogisticRegressionWithLI
import org.apache.spark.mllib.classification.LogisticRegressionModel
import org.apache.spark.mllib.evaluation.MulticlassMetrics
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.util.MLUtils

// Load training data in LIBSVM format.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data

// Split data into training (60%) and test (40%).
val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0).cache()
val test = splits(1)

// Run training algorithm to build the model
val model = new LogisticRegressionWithLBFGS()
  .setNumClasses(10)
  .run(training)

// Compute raw scores on the test set.
val predictionAndLabels = test.map {
  case LabeledPoint(label, features) =>
    val prediction = model.predict(features)
    (prediction, label)
}

// Get evaluation metrics.
val metrics = new MulticlassMetrics(predictionAndLabels)
val precision = metrics.precision
println("Precision = '' + precision)

// Save and load model
model.save(sc, "myModelPath")
val sameModel = LogisticRegressionModel.load(sc, "myModelPath")
```

Linear Regression

Linear Regression

Linear regression is a simple yet useful learning algorithms, which can be regarded as a statistical problem or an optimisation problem. For simple linear regression there is, in fact, an optimal solution, however for high dimensional spaces there are none and we need to regard the problem as an optimisation problem. Regression fits a function to the data set, so what we are trying to do is to find a representative function and fit to our data set. Learning takes place as finding the best possible - *local optimum* -values of the function parameters. Linearity refer to the fact that we are tying to fit either a straight line or a polynomial function (polynomial regression).

In this chapter we will go through the statistical account and the optimisation account. The latter is more relevant to machine learning.

Statistical Simple Linear Regression

Statistical Simple Linear Regression

Suppose we have a data sample which is pro- duced from a single variable such that $y = f(x)$. As we do not know which f has produced the data we need to infer and predict y given x from the sample. We assume that the function is linear so that a straight line can be fitted to the data and that there is a noise component (the *residual* or *error*), u, so that our predictions from the linear model are inexact. Let i be the index of sample data points (x_i, y_i), then the simplest linear model is given by

$$y_i = \alpha + \beta x_i + u_i$$

where α is the y-intercept and β is the slope.

In order to make use of this model, it has to be further constrained by the following constraints

$$\begin{aligned} E(u_i) = & \quad 0 \\ E(u_i^2) = & \quad \sigma_i^2 \\ E(u_i, u_j) = & \quad 0, \\ & \quad i \neq j \end{aligned}$$

which means: 1) the mean of u is 0 for all i, the variance of u is the same for all i and there is no correlation across observations. We should show that our model possesses these properties later on.

Our task for now is to estimate the value of y as an estimation of the values of the coefficients α and β, so that the estimated output value, $\hat{y}$ is given by

$$\hat{y}_i = \hat{\alpha} + \hat{\beta} x_i$$

The residual may be estimated as follows

$$\begin{aligned} \hat{u}_i & = y_i - \hat{y}_i \\ & = y_i - (\hat{\alpha} + \hat{\beta} x_i) \end{aligned}$$

which is the difference distance between an actual observation and the fitted line. This is an estimation because we employ estimation of the parameters α and β. For a good fit we need to have small residuals. At this point, we have to choose criteria for the best line. Let $\{-5, 2, 3\}, \{-1, 2, -1\}, \{-2, 2, 0\}$ be three residual sets for three lines fitted to the same sample, we investigate different criteria on these points.

1. The line which gives the smallest *sum of residuals*

$$min \mid \sum_{i=1}^{n} (y_i - \hat{y}_i) \mid$$

in this case we have

$$\begin{aligned} -5 + 2 + 3 = &\quad 0 \\ -1 + 2 - 1 = &\quad 0 \\ -2 + 2 + 0 = &\quad 0 \end{aligned}$$

which makes it clear that positive and negative residuals over the whole sample cancel out. That is, all lines are equal.

1. The line which give the *smallest sum of absolute deviations*

$$min \sum_{i=1}^{n} |(y_i - \hat{y}_i)|$$

then we have the following differences across the three lines

$$\begin{aligned} 5 + 2 + 3 = &\quad 10 \\ 1 + 2 + 1 = &\quad 4 \\ 2 + 2 + 0 = &\quad 4 \end{aligned}$$

which is slightly better than the sum of residuals but lines are not always distinguishable, also, this is computationally intensive.

1. The line which gives the *smallest sum of squared residuals, SSR, or ordinary least squares, OLS*

$$min \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

then we have

$$\begin{aligned} 25 + 4 + 9 = &\quad 38 \\ 1 + 4 + 1 = &\quad 6 \\ 4 + 4 + 0 = &\quad 8 \end{aligned}$$

which weighs outliers well, efficient and unbiased.

Clearly, we should choose the smallest sum of squared residuals, SSR. Hence, we need to minimise the following equation

$$SSR = \sum_{i=1}^{n} u^2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} (y_i - \hat{\alpha} - \hat{\beta} x_i)^2$$

In order to minimise this equation, according to calculus, we need to partially differentiate the equation with respect to $\hat{\alpha}$ and with respect to $\hat{\beta}$, setting both equations to zero, which produces two equations in two unknowns (the normal equations of least square).

$$\frac{\partial SSR}{\partial \hat{\alpha}} = -2 \sum_{i=1}^{n} y_i - \hat{\alpha} - \hat{\beta} x_i = 0 \qquad (1)$$

$$\frac{\partial SSR}{\partial \hat{\beta}} = -2 \sum_{i=1}^{n} x_i(y_i - \hat{\alpha} - \hat{\beta} x_i) = 0 \qquad (2)$$

Eq (1) implies

$$\begin{aligned} \sum_{i=1}^{n} y_i - n \cdot \hat{\alpha} - \hat{\beta} \cdot \sum_{i=1}^{n} x_i &= 0 \\ \hat{\alpha} &= \frac{1}{n} \sum_{i=1}^{n} y_i - \hat{\beta} \frac{1}{n} \sum_{i=1}^{n} x_i \qquad (3) \\ \hat{\alpha} &= \bar{y} - \hat{\beta} \bar{x} \end{aligned}$$

where $\bar{y}$ is the sample mean of y and $\bar{x}$ is the sample mean of x.

Eq (2) implies

$$(\sum_{i=1}^{n} x_i)\, y_i + \hat{\alpha} \sum_{i=1}^{n} x_i - \hat{\beta} \sum_{i=1}^{n} x_i^2 = 0 \qquad (4)$$

Substituting (3) in (4)

$$\begin{aligned} (\sum_{i=1}^{n} x_i)\, y_i + (\bar{y} - \hat{\beta} \bar{x}) \sum_{i=1}^{n} x_i - \hat{\beta} \sum_{i=1}^{n} x_i^2 &= 0 \\ (\sum_{i=1}^{n} x_i)\, y_i + \bar{y} \sum_{i=1}^{n} x_i - \hat{\beta} \bar{x} \sum_{i=1}^{n} x_i - \hat{\beta} \sum_{i=1}^{n} x_i^2 &= 0 \\ (\sum_{i=1}^{n} x_i)\, y_i + \bar{y} \sum_{i=1}^{n} x_i - \hat{\beta} (\sum_{i=1}^{n} x_i^2 - \bar{x} \sum_{i=1}^{n} x_i) &= 0 \end{aligned}$$

$$\hat{\beta} = \frac{(\sum_{i=1}^{n} x_i)\, y_i + \bar{y} \sum_{i=1}^{n} x_i}{\sum_{i=1}^{n} x_i^2 - \bar{x} \sum_{i=1}^{n} x_i} \qquad (5)$$

However, if you look at (5), although mathematically correct, it is rather hard to interpret. There is another way to derive $\hat{\beta}$ in a more meaningful as well as an easier way

$$\begin{aligned} \hat{u}_i &= y_i - \hat{\alpha} - \hat{\beta} x_i \\ &= \bar{y} - \hat{\beta} \bar{x} \\ &= (y_i - \bar{y}) - \hat{\beta} (x_i - \bar{x}) \\ &= \mathbf{y}_i - \hat{\beta} \mathbf{x}_i \end{aligned}$$

where $\mathbf{y}_i$ and $\mathbf{x}_i$ are the deviations from the sample means.

Hence,

$$SSR = \sum_{i=1}^{n} (\mathbf{y}_i - \hat{\beta} \mathbf{x}_i)^2$$

Therefore,

$$\begin{aligned}
\frac{\partial SSR}{\partial \hat{\beta}} = -2\sum_{i=1}^{n}(\mathbf{y}_i - \hat{\beta}\mathbf{x}_i)\mathbf{x}_i &= 0 \\
-\sum_{i=1}^{n}\mathbf{y}_i\mathbf{x}_i + \hat{\beta}\sum_{i=1}^{n}\mathbf{x}_i^n &= 0 \\
\hat{\beta}\sum_{i=1}^{n}\mathbf{x}_i^n &= \sum_{i=1}^{n}\mathbf{y}_i\mathbf{x}_i \\
\hat{\beta} &= \frac{\sum_{i=1}^{n}\mathbf{y}_i\mathbf{x}_i}{\sum_{i=1}^{n}\mathbf{x}_i} \\
\hat{\beta} &= \frac{cov(y_i, x_i)}{var(x_i)}
\end{aligned}$$

Constraints on OLS

1. The OLS line passes through the midpoint of data sample. To prove this, evaluate the $\hat{y}$ at the sample mean $\bar{x}$

$$\hat{y} = \hat{\alpha} + \hat{\beta}\bar{x}$$

from (3)

$$\begin{aligned}
\hat{\alpha} + \hat{\beta}\bar{x} &= (\bar{y} - \hat{\beta}\bar{x}) + \hat{\beta}\bar{x} \\
&= \bar{y}
\end{aligned}$$

1. The mean of the residual is zero

$$\begin{aligned}
\bar{u}_i &= \frac{1}{n}\sum u_i \\
&= \frac{1}{n}\sum(y_i - \hat{y}_i) \\
&= \frac{1}{n}\sum y_i - \frac{1}{n}\sum(\hat{\alpha} + \hat{\beta}\bar{x}^n) \\
&= \bar{y} - \hat{\alpha} + \hat{\beta}\bar{x} \\
&= \bar{y} - \bar{y} \\
&= 0
\end{aligned}$$

1. Residuals are uncorrelated with the input variable so that the fitted line can be imporved by changing the slope.

$$
\begin{aligned}
cov(y_i, x_i) &= \frac{1}{n} \sum u_i x_i \\
&= \frac{1}{n} \sum (\mathbf{y}_i - \hat{\beta} \mathbf{x}_i) x_i \\
&= \frac{1}{n} \sum y_i x_i - \hat{\beta} \frac{1}{n} \sum x_i^2 \\
&= cov(y_i, x_i) - \hat{\beta}\ var(x_i) \\
&= cov(y_i, x_i) - \frac{cov(y_i, x_i)}{var(x_i)}\ var(x_i) \\
&= 0
\end{aligned}
$$

Optimisation

Optimisation

Let's go through an example. Given a set of house sizes and house prices such that the data set is (x_i, y_i) where $x_i$is the house size and $y_i$is the house price. The task is to estimate the house price given a house size. If we regard the house size and the target price as continous, we can model the situation as a regression problem. Because we want to fit a linear function to the data or approximate the output as a linear function of the output this can be modelled as a linear regression problem. The data set has holes meaning that it does not contain the size and price of every single house so the task is to estimate the target price given an arbitrary house size.

The model pipeline, as a supervised learning problem, is *training set* $\longrightarrow$ *learning algorithm* $\longrightarrow$ *hypothesis*. The training set is fed into the learning algorithm which outputs a function, conventionally, called the hypothesis and denoted by h. h is a function mapping from input to output or features to output such that $h : x \longrightarrow y$. The first task in solving this problem is to decide what representation of h should be used. In this certain problem we will use an affine function, such that,

$$h = h_\theta = \theta_0 + \theta_1 x$$

where h_θ is a way to assert that we are estimating θ. In the more general case, we might have an n number of features so that if we assume a constant input $x_0 = 1$

$$h = \theta_0 + \theta_1 x_1 + \cdots + \theta_{n-1} x_{n-1}$$

which could be more compactly written as

$$h = \sum_{j=0}^{n} \theta_j x_j$$

and if we regard θ and x as vectors such that $(\theta_0, \theta_1, \cdots, \theta_n)$ and $(x_0, x_1, \cdots, x_n)$ then we can define h as follows

$$h = \sum_{j=0}^{n} \theta_j x_j = \theta^T x$$

which opens the way to more efficient vectorised implementations using linear algebra libraries. The constant parameter $x_0 = 1$ is called the *bias*. Depending on your background it could be differently regraded but the important point here is that it allows for efficient implementation (also bias value could be adjusted to shift the approximated function, which is sometimes required).

As we see θ is the vector of parameters to the learning algorithm. The training task is to set the

learning parameters in order to make predications. One way of setting the parameters is to iteratively minimise the difference between predicated outputs of the algorithm and the actual output in the training set in each iteration. The difference in this case is the error in predication. The function that measures this error is called the *cost function*. Let m be the number of training examples. The cost function is then

$$min_\theta \frac{1}{m} \sum_{i=1}^{m} (h(x_i), y_i)^2$$

which means we minimise the the total average error between predications and actual output at all iterations by adjusting the parameters vector θ . This equation is called the *mean square error*, MSE, so we are minimising MSE over θ . Let $J(\theta)$ be the cost function. Because we are going to differentiate this function in a later step, mathematics will be simplified if we minimise half of the MSE instead - which should have the same effect - so that the we are going to minimise the cost function as such

$$min_\theta J(\theta) \equiv min_\theta \frac{1}{2} \frac{1}{m} \sum_{i=1}^{m} (h(x_i), y_i)^2$$

Gradient Descent

Gradient Descent

So how should we minimise the cost function? If we imagine the the surface of the cost function as a hill and we are standing on a certain point on the hill, then because we don't know about the whole hill but only the point we are at, what we need to do is to take a step in the downward steepest direction at each iteration hoping that we reach bottom. This method does not guarantee reaching the bottom if there are multiple local minima — points on the hill where every other point around is higher but there are lower points other places on the hill.

Gradient descent does exactly that. It is an algorithm that is not guaranteed to converge to a global minimum or even to the best local minimum but it usually works taking some considerations into account.

Intuitively, we can differentiate the cost function to calculate the slope and then we move to the estimated output in the opposite direction which should be the steepest downward direction. To achieve this we need a step size, η. So at each step we adjust the parameters as such

$$\theta = \theta - \eta \frac{d}{d\theta} J(\theta)$$

but the parameter is not a scalar, it is a vector and we need to individually adjust the parameters in order to minimise the contribution of each parameter to the total error. So we need to partially differentiate the cost function in respect to each parameter separately and adjust that parameter accordingly

$$\theta_j = \theta_j - \eta \frac{\partial}{\partial \theta_j} J(\theta), \quad j \in [0, n-1]$$

note that we are subtracting the partial derivative because when the slope is non-negative we need to go on the opposite direction and when the slope negative we need to add it to go in the downward direction. The step size η is called the learning rate. η controls how far the estimation moves at each step. It is a very important parameter because if it is too small, unnecessarily more iterations will be needed to reach the minimum and if it is too large, the minimum could be easily skipped over so that the algorithm oscillates that it even could not eventually converge — too large η could cause the algorithm to diverge.

Another important point is that gradient descent can converge even with a fixed learning rate. The learning rate is multiplied by the slope so that algorithm adapts the step size accordingly. In an idealised situation, the step size will begin largest as the estimation worst and will be smaller at each step as the estimation is re ned. This is because the steepness of the slope would decrease as we are reaching a minimum. At the minimum the slope will be zero and convergence is reached. This should be detected by absence of parameters updates. Gradient descent, thus, could be summarised as follows:

Repeat until convergence: compute all $\eta \ \frac{\partial}{\partial \theta_j} J(\theta)$ then update all θ_j.

To compute the gradient descent at each step we need to substitute the de nition of the cost function.

Let the gradient be G

$$G = \frac{\partial}{\partial \theta_j} J(\theta)$$

by chain rule

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{\partial J(\theta)}{\partial y^{(i)}} \frac{\partial y^{(i)}}{\partial \theta_j}$$

$$= \frac{\partial \frac{1}{m} \sum_{i=1}^{m} (h(x^{(i)}), y^{(i)}))^2}{\partial y^{(i)}} \cdot \frac{\partial \sum_{j=1}^{n} \theta_j x_j^{(i)}}{\partial \theta_j}$$

$$= \frac{1}{2m} \cdot 2 \cdot \sum_{i=1}^{m} (h(x^{(i)} - y^{(i)}) \cdot -1 \cdot x_j^{(i)}$$

$$= -\frac{1}{m} \sum_{i=1} m(h(x^{(i)} - y^{(i)}) \cdot x^{(i)}$$

however, deriving the differential in this manner gives the upward steepest direction so we have to go to the opposite direction, hence we define the differential as

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1} m(h(x^{(i)} - y^{(i)}) \cdot x^{(i)}$$

and we compute the update as follows

$$\hat{\theta}_j = \theta_j - \eta \, \frac{1}{m} \sum_{i=1} m(h(x^{(i)} - y^{(i)}) \cdot x^{(i)}$$

note that if $\frac{1}{m} \sum_{i=1} m(h(x^{(i)} - y^{(i)})$ could be calculated in parallel then all θ_j could be calculated as once. This is why the algorithm we have derived is called *batch gradient descent*.

For completeness, you should note that we had to invert the sign of the di erential for the reason that the di erential is the upward steepest direction at the point we are standing on the hill. This has to do with the de nition of the gradient as the upward steepest direction. You may want to look at the mathematical appendix in case you want to know how the gradient is mathematically de ned.

So the question which should be raised by now is how could we manage to notice whether the algorithm is going to converge and whether there is any guarantee of convergence at all? It turns out that de ning gradient descent as we had de ned it for linear regression will always converge given the implementation is bug-free and that the learning rate is appropriate. This is because, mathematically, the cost function we are using will always have a global minimum and no local minima regardless of

the number of learning parameters θ_j. Then the question reduces to how to set the learning rate. There is no known method to guarantee choosing a value for the learning rate that will make the algorithm converge but there is a heuristic method. Start with a small value such as 0.001 and keep multiplying it by a small constant such as 3 until the algorithm diverges or its performance degrades, then the previous value of η is the best value to set.

Implementation

Linear Regression Implementation

In this section we will implement two variants of the linear regression gradient descent algorithm. The two implementations differ in a slight detail, however, it is important to appreciate such a detail. We also will highlight some characteristics of the algorithm. After going through this section, hopefully, you will appreciate giving attention to real-life computer limitations and how they differ from mathematics.

The dataset used is for the list price and the best price for GMC (reference Consumer's Digest).

```
from operator import add, mul

def process_training_set(examples):
    result = {}
    for k, v in examples.iteritems():
        result[(1, k)] = v
    return result

def estimate(x, params):
    return hypothesis((1, x), params)

def hypothesis(x, params):
    return sum(map(mul, x, params))

def delta(training_set, params, previous_cost, learning_rate):
    current_cost = 0
    for x, y in training_set.iteritems():
        current_cost = current_cost + hypothesis(x, params) - y
    current_cost = (1.0 / len(training_set)) * current_cost
    if current_cost == previous_cost: return True, current_cost
    for j in range(len(params)):
        params[j] = params[j] - (learning_rate * current_cost)
    return False, current_cost

def delta2(training_set, params, previous_cost, learning_rate):
    cost = 0
    for x, y in training_set.iteritems():
        cost = cost + hypothesis(x, params) - y
        cost = (1.0 / len(training_set)) * cost
    for x in training_set.iterkeys():
        for j in range(len(params)):
            params[j] = params[j] - (learning_rate * cost * x[j])
        if cost == previous_cost: return True, cost
    return False, cost

def train(training_set, params, learning_rate=0.1, delta=delta):
    result, c = delta(training_set, params, float('nan'), learning_ra
    while not result:
```

```
            result, c = delta(training_set, params, c, learning_rate)
        print 'params:', params, 'cost-funtion:', c
        return params, c

def estimate_set(data_set, params):
        s = ((1, x) for x in data_set)
        result = {}
        for x in s:
            result[x[1]] = hypothesis(x, params)
        return result

def estimation_mse(reference, estimates):
        mse = 0
        for k, v in reference.iteritems():
            mse = mse + (estimates[k] - v) ** 2
        return len(reference) * mse

training_set = {
        12.39999962: 11.19999981,
        14.30000019: 12.5,
        14.5:        12.69999981,
        14.89999962: 13.10000038,
        16.10000038: 14.10000038,
        16.89999962: 14.80000019,
        16.5:        14.39999962,
        15.39999962: 13.39999962,
        17:          14.89999962,
        17.89999962: 15.60000038,
        18.79999924: 16.39999962,
        20.29999924: 17.70000076,
        22.39999962: 19.60000038,
        19.39999962: 16.89999962,
}

test_set = {
        15.5:        14,
        16.70000076: 14.60000038,
        17.29999924: 15.10000038,
        18.39999962: 16.10000038,
        19.20000076: 16.79999924,
        17.39999962: 15.19999981,
        19.5:        17,
        19.70000076: 17.20000076,
        21.20000076: 18.60000038,
}
```

The `train` function takes a parameter function to carry out the real training. There are two implementations provided, `delta` and `delta2`. `delta2` adheres faithfully to the mathematical definition above. However, `delta` does something interesting. Looking at `delta` definition drops the multiplication by $x_j^{(i)}$ when updating the parameters. In effect, `delta` does not compute the full gradient as a contribution of the error at the output of every parameter, it computes the the derivative of the cost function in respect to the output and subtracts it from each parameter. In this specific case,

the difference between the two implementations is rather very small, however, it might make a difference to other applications. Nonetheless, the approach of `delta` is rather widely implemented and works although it doesn't follow the mathematical definition. The `delta` implementation is more efficient because it avoids calculating the cost function at each iteration, rather it calculates it once for the whole dataset in order to achieve adjustment of all parameters for once. Also, this approach lends itself more efficiently to parallel execution and map-reduce implementations.

Let's now assess the characteristics of the algorithm. An important issue is the limitation of the floating-point representation. To attain a strictly zero derivative might require impractically slow floating-point representations. Hence, other conditions of convergence are required. The parameters vectors could be inspected to be the same on two consecutive iterations or the difference between the current vector and the previous could be required to be within a very small threshold. The same approach could be used with the value of the cost function. In our implementation the cost function was required to be same in two consecutive iterations in order to stop the training.

Vectorised Implementation

```
from operator import add, mul
import numpy as np

def process_training_set(examples):
    result = []
    for k, v in examples.iteritems():
        result.append((np.array((1, k)), v))
    return result

def estimate(x, params):
    return hypothesis(np.array((1, x)), params)

def hypothesis(x, params):
    return np.dot(x, params)

def delta(training_set, params, previous_cost, learning_rate):
    current_cost = 0
    for t in training_set:
        current_cost = current_cost + hypothesis(t[0], params) - t[1
    current_cost = (1.0 / len(training_set)) * current_cost
    if current_cost == previous_cost: return True, current_cost, para
    for j in range(len(params)):
        params[j] = params[j] - (learning_rate * current_cost)
    return False, current_cost, params

def delta2(training_set, params, previous_cost, learning_rate):
    cost = 0
    for t in training_set:
        cost = cost + hypothesis(t[0], params) - t[1]
        cost = (1.0 / len(training_set)) * cost
        if cost == previous_cost: return True, cost, params
    for t in training_set:
        params = params - (learning_rate * cost * t[0])
    return False, cost, params

def train(training_set, params, learning_rate=0.1, delta=delta):
```

```
    result, c, params = delta(training_set, params, float('nan'), lea
    while not result:
        result, c, params = delta(training_set, params, c, learning_
    print 'params:', params, 'cost-funtion:', c
    return params, c

def estimate_set(data_set, params):
    s = (np.array((1, x)) for x in data_set)
    result = {}
    for x in s:
        result[x[1]] = hypothesis(x, params)
    return result

def estimation_mse(reference, estimates):
    mse = 0
    for k, v in reference.iteritems():
        mse = mse + (estimates[k] - v) ** 2
    return len(reference) * mse

training_set = {
    12.39999962: 11.19999981,
    14.30000019: 12.5,
    14.5:        12.69999981,
    14.89999962: 13.10000038,
    16.10000038: 14.10000038,
    16.89999962: 14.80000019,
    16.5:        14.39999962,
    15.39999962: 13.39999962,
    17:          14.89999962,
    17.89999962: 15.60000038,
    18.79999924: 16.39999962,
    20.29999924: 17.70000076,
    22.39999962: 19.60000038,
    19.39999962: 16.89999962,
}

test_set = {
    15.5:        14,
    16.70000076: 14.60000038,
    17.29999924: 15.10000038,
    18.39999962: 16.10000038,
    19.20000076: 16.79999924,
    17.39999962: 15.19999981,
    19.5:        17,
    19.70000076: 17.20000076,
    21.20000076: 18.60000038,
}
```

Spark Implementation

Linear Regression Spark Implementation

```
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.regression.LinearRegressionModel
import org.apache.spark.mllib.regression.LinearRegressionWithSGD
import org.apache.spark.mllib.linalg.Vectors

// Load and parse the data
val data = sc.textFile("data/mllib/ridge-data/lpsa.data")

val parsedData = data.map { line =>
  val parts = line.split(',')
  LabeledPoint(
    parts(0).toDouble,
    Vectors.dense(parts(1).split(' ').map(_.toDouble)))
}.cache()

// Building the model
val numIterations = 100
val stepSize = 0.00000001
val model = LinearRegressionWithSGD.train(parsedData, numIterations,

// Evaluate model on training examples and compute training error
val valuesAndPreds = parsedData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}

val MSE = valuesAndPreds.map{case(v, p) => math.pow((v - p), 2)}.mea
println("training Mean Squared Error = " + MSE)

// Save and load model
model.save(sc, "myModelPath")
val sameModel = LinearRegressionModel.load(sc, "myModelPath")
```

Multilayer Perceptron

Multilayer Perceptron

```
import org.apache.spark.ml.classification.MultilayerPerceptronClassi
import org.apache.spark.ml.evaluation.MulticlassClassificationEvalua

// Load the data stored in LIBSVM format as a DataFrame.
val data = sqlContext.read.format("libsvm")
  .load("data/mllib/sample_multiclass_classification_data.txt")

// Split the data into train and test
val splits = data.randomSplit(Array(0.6, 0.4), seed = 1234L)
val train = splits(0)
val test = splits(1)

// specify layers for the neural network:
// input layer of size 4 (features), two intermediate of size 5 and 
// and output of size 3 (classes)
val layers = Array[Int](4, 5, 4, 3)

// create the trainer and set its parameters
val trainer = new MultilayerPerceptronClassifier()
  .setLayers(layers)
  .setBlockSize(128)
  .setSeed(1234L)
  .setMaxIter(100)

// train the model
val model = trainer.fit(train)

// compute precision on the test set
val result = model.transform(test)
val predictionAndLabels = result.select("prediction", "label")

val evaluator = new MulticlassClassificationEvaluator()
  .setMetricName("precision")
println("Precision:" + evaluator.evaluate(predictionAndLabels))
```