ASSIGNMENT A

Date: 28/02/2018

Module: Programming for Big Data

Student: Pierluca Del Buono

Course code: DT228B Data Analytics Part Time Year 2

Student Number: D15128655

Lecturer: Brendan Tierney

Contents

1. Loading Process Overview

In the present work, three languages have been compared: Latin, Italian and Romanian. Latin is the father language of all the romance linguistic family, to which Italian and Romanian belong; the purpose was to study whether letter distribution in Latin has been retained in the language that have it as ancestor.

Files related to books written in these idioms have been downloaded from Gutenberg project site, and placed in three separated folders under a root folder in the local host, called Books, to be processed by MapReduce jobs.

A shell script has been put together in order to:

- read the input files from a folder in the localhost
- iterate through its subfolder
- put them on *HDFS*.
- process them via Map Reduce jobs.
- get the results merged back to the local host.

As a side note, the input folder is accepted as the only parameter of the script, the script will iterate across its subfolders, regardless of their number, and will launch for them the same *MapReduce* job. The script is fully scalable, in the sense that can manage any number of books within the folder.
The results of the computation will be merged back then to the local host, creating an output folder. Merge is needed because of the partitioning phase, that will lead to more than 1 output files for language.

Specifically, to this case, each folder was storing 3 books; to start the process, the following command has been issued:

bash ~/letter_frequency.sh ~/book

bash shell script letter_frequency.sh is contained in the snippet below.

```
#removing existing books folder
hdfs dfs -rm -R /books/*

#creating folder in HDFS as in source
for folder in $@/*
 do
   hdfs dfs -mkdir /books/`basename $folder`
 done

#copying files on HDFS
for folder in $@/*
 do
```

```
   hdfs dfs -put $folder/* /books/`basename $folder`/
 done

#executing Hadoop map reduce processes
for folder in `hdfs dfs -ls /books | awk '{print $8}'`
 do
    hdfs dfs -rm -R $folder/output
    hadoop jar LetterFrequency.jar LetterFrequency $folder $folder/output
 done

#merging hadoop output back to source
for folder in `hdfs dfs -ls /books/*/ | grep "^d" | awk '{print $8}'`
  do
    dirfolder=`dirname $folder`
    hadoop dfs -getmerge $folder $@/`basename $dirfolder`out.txt
  done
```

2. Process Design

In this assignment 3 languages were analysed with 3 books each:

Latin:

- Marcus Tullius Cicero - Orationes
- Gaius Valerius Catullus - Carmina
- Caius Julius Cesar - De Bello Gallico

Italian:

- Carlo Collodi – Pinocchio
- Dante Alighieri – La Divina Commedia
- Marco Polo – Il Milione

Romanian:

- Teodor Dorin Moisa – Creierul, o Enigma Descifrata
- Mihai Eminescu – Poezii
- Eugen Ionescu – Chantareata Cheala

For this specific assignment, a cluster with a NameNode and a single DataNode has been used, but the process can scale up adding more DataNodes, that will share among them the *Mapper* and *Reduce* tasks.
The HDFS filesystem used in the work had a block size set to the default, 128 MB, and a factor of replication equals to 3, that is the number of replicas managed by the DataNode on input of the NameNode, for fault-tolerance.

In the picture at the end of the paragraph have been represented 3 Mappers, however, the number of the map tasks is determined by the number of the files to process divided by the dimension of the splits. Is good practice to keep the dimension of the splits of the same size of the HDFS block, in this case 128 MB. The Mapper tasks then are evenly distributed across the DataNodes, and the NameNode assign possibly a task to a split local to a node.
In the case of the present assignment, 3 Mappers Tasks were spawn in each MapReduce jobs, as the number of the file for each language. A dedicated MapReduce Process was executed for each language.

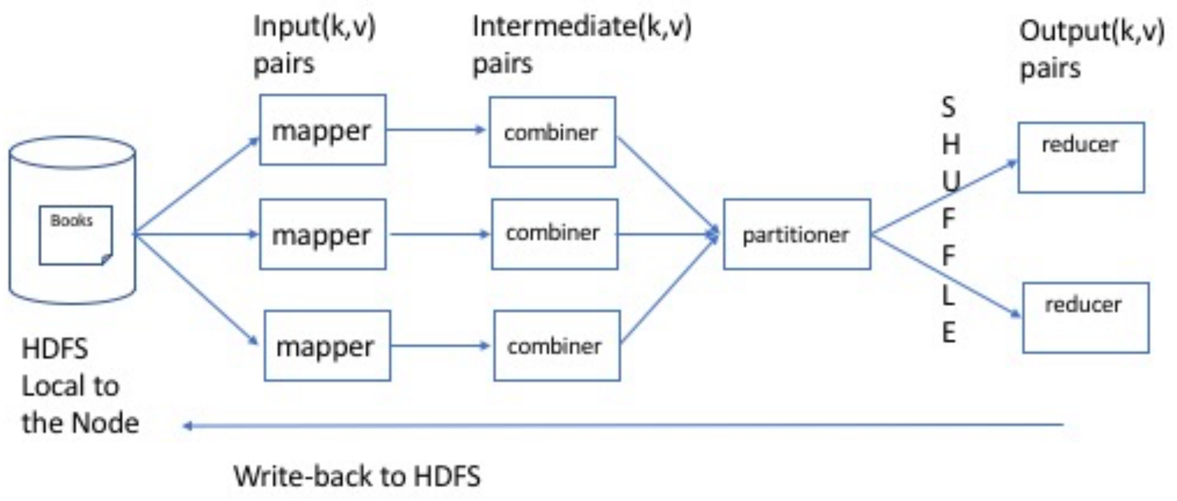
Is important to notice that in picture have been not represented some passage that are between HDFS and the Mapper, and they are:

InputFormat: Selection of files from HDFS to be input of the mapper.
InputSplit: Splitting of files in chunk of fixed dimension, executed by the InputFormat Class.
RecordReader: Presentation of the byte output to the mapper in form of records.

Mapper tasks will work to map the records in input to pairs of key-values, where key will be a letter composing the record, and value will be initially 1.0; inside each Mapper task will be defined a counter that keep track of all the letter processed, such counter will be used then in reducing phase, to pull out a frequency.

The output of each mapper will be redirected to a *Combiner*, process that usually is defined by the same code of the reducers, but is running directly on the output of the *Mappers*.
In this case, it will be slightly different from the *Reducer* step, since it will not use the counter defined in the *Mapper*, like the *Reducer* will do.

The number of *Reducers* tasks needs to be set up beforehand, via the setReducenum() method in the driver. Is important to choose that number in a way that the *Reducers* will not process an excessive amount of data, and the share of data across them is similar, to avoid bottleneck. Since the distribution of the vowel and the consonant appeared to be quite similar in the language processed, a *Partitioner* has been defined to split the data into vowel and consonant, and to send them to a distinct reducer. Is interesting to notice that this configuration is more useful when scaling up: most of the vendor suggest to not have a *Reducer* processing less than 1 Gb of data, because of the overhead caused spawning those tasks.
Data will be sorted and Shuffled before being processed by each *Reducer*.



The picture above represents the MapReduce process and its main tasks. Files are extracted by HDFS and written back there, once terminated via the *OutputFormat* Class.

3. Map Reduce Code

3.1 Driver

The Driver is the entry point for the whole program, because it defines all the other components. The class for the driver is LetterFrequency, and its main method accepts input and output path. The job gets named, and the number of reducers is set to 2, because of technical decision. In the Driver are defined also the classes for the *Mapper*, the *Combiner*, The *Partitioner* and The *Reducer*. At the end of the driver is specified the output format of the Reducer that is written in the output path.

```
import org.apache.hadoop.fs.Path;
//DoubleWritable class used in place of IntWritable, to take into account
decimals
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class LetterFrequency {
    public static void main(String[] args) throws Exception {

        if (args.length != 2) {
            System.err.println(args);
            System.err.println("Usage: LetterFrequency <input path> <output
path>");
```

```
            System.exit(-1); }

        System.out.println("In Driver now!");

        Job job = Job.getInstance();
        job.setJarByClass(LetterFrequency.class);
        job.setJobName("LetterFrequency");

        // set number of reducers, change to 0 to control mapper output.
        job.setNumReduceTasks(2);

        job.setMapperClass(LetterMapper.class);

        job.setCombinerClass(FrequencyCombiner.class);

        job.setPartitionerClass(LetterPartitioner.class);

        job.setReducerClass(FrequencyReducer.class);

        // output set to use DoubleWritable
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(DoubleWritable.class);

        //using paths provided as argument
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

3.2 Mapper

The mapper section of the code takes as input the source file in the input path, in the form of records. Within the class LetterMapper, that extends Mapper class, is defined a counter that will be incremented for every letter passed to the combiner.
The *Mapper* in its map method will Iterate through all the letter of all the words of the input, and using the utility class isLetter, will pass the key-value pair letter, 1.0 to the *Combiner*.

```
import java.io.IOException;

import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
```

```
// extending Mapper Class
public class LetterMapper extends Mapper<LongWritable, Text, Text,
DoubleWritable> {

    // Static Counter will count the number of letter going through the
reducers in total.
    static enum MapperCounter {
        MAPPER_RECORD_COUNTER
    }

    public void map(LongWritable key, Text value, Context context)
                    throws IOException, InterruptedException {

        //lowercase of the string
        String s = value.toString().toLowerCase();

        // going through all the words composing text
        for (String word : s.split("\\W+")) {
          // going through all the letters composing a word
          for (int i = 0; i < word.length(); ++i) {
            char character = word.charAt(i);
            // using isLetter function to process letter
            if (isLetter(character)) {
              // incrementing counter

context.getCounter(MapperCounter.MAPPER_RECORD_COUNTER).increment(1);
              // send letter to combiner
              context.write(new Text(String.valueOf(character)), new
DoubleWritable(1.0));
            }
          }
   }
}
    // utility class returning TRUE for letters
    private boolean isLetter(char character) {
        return (65 <= character && character <= 90)
                || (97 <= character && character <= 122);
    }
}
```

3.4 Combiner

FrequencyCombiner Class will run as a mini-reducer on the output of the *Mapper*, reducing the intermediate data that will be passed to the reducer. Its code is differing from the Reducer code, where the final value gets divided by the Total Count value, to return a frequency.

```
import java.io.IOException;

import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class FrequencyCombiner extends Reducer<Text, DoubleWritable, Text,
DoubleWritable> {

    public void reduce(Text key, Iterable<DoubleWritable> values, Context
context)
            throws IOException, InterruptedException {
      // initialating letterCount
      double letterCount = 0.0;
      System.out.println(" In Combiner now!");
      for (DoubleWritable value : values) {
        // incrementing letterCount
        letterCount += value.get();
      }
      // send combiner output to reducer
      context.write(key, new DoubleWritable(letterCount));
    }
}
```

3.5 Partitioner

Partition LetterPartitioner redirects key-value pairs to one of the two reducers, one assigned to vowels and one assigned to consonants
.

```
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;

public class LetterPartitioner extends Partitioner<Text, DoubleWritable> {

  //overriding getpartition method
  @Override
  public int getPartition(Text key, DoubleWritable value, int numReduceTasks)
{

    String letter = key.toString();

    // sending vowel to a reducer
    if ("aeiou".indexOf(letter) < 0) {
      return 0;
    }
    else {
      // sending consonant to the second reducer
      return 1;
    }
  }
}
```

3.6 Reducer

FrequencyReducer extends the reducer class, reducing input key/values assigned by the Partitioner from the Combiner. Within Reducer class is overridden the method setup, to retrieve the counter defined in the mapper class, counter for all the letter processed.
At the end said counter will be used to divide the values.

```
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Cluster;
import org.apache.hadoop.mapreduce.Reducer;

// Frequency Reducer extending reducer class
public class FrequencyReducer extends Reducer<Text, DoubleWritable, Text,
DoubleWritable> {

  // declaring totalCount that will bear total number of letter for the
process
  double totalCount;

  //overriding set up method to initialite a cluster configuration, to
retrieve the total counter
  @Override
  public void setup(Context context) throws IOException,
InterruptedException{
    // getting configuration
    Configuration conf = context.getConfiguration();
    // initialiating cluster
    Cluster cluster = new Cluster(conf);
    // getting current job settings
    org.apache.hadoop.mapreduce.Job currentJob =
cluster.getJob(context.getJobID());
    // get MAPPER_RECORD_COUNTER value
    totalCount =
currentJob.getCounters().findCounter(LetterMapper.MapperCounter.MAPPER_RECORD
_COUNTER).getValue();
```

```
  }

  public void reduce(Text key, Iterable<DoubleWritable> values, Context
context) throws IOException, InterruptedException {
    // initialiating letterCount
    double letterCount = 0.0;
    double frequency;
    System.out.println(" In Reducer now!");
    for (DoubleWritable value : values) {
      // incrementing letterCount
      letterCount += value.get();
    }
    // calculating frequency using totalCount
    frequency = letterCount/totalCount;
    // writing Frequency in the reducer output
    context.write(key, new DoubleWritable(frequency));
  }
}
```

4. Output Files

Below an extract of the output file coming from a *Mapper*, before it gets worked by the combiner: each key has the value of 1, expressed with a *Double*, since we are interested in decimal as final output.

```
t       1.0
h       1.0
e       1.0
p       1.0
r       1.0
o       1.0
j       1.0
e       1.0
c       1.0
t       1.0
g       1.0
u       1.0
t       1.0
e       1.0
n       1.0
b       1.0
e       1.0
```

The dataset then goes through a combiner, that is local to the mapper, where values are pre-reduced by their keys.

Below the output of *Reducers*, the final task, in the case of Italian, after the merge of the files in the local host.

```
a 0.11098100393420458
b 0.010066856220702374
c 0.04846686395906406
d 0.037542339317487246
e 0.1171899644196578
f 0.011635486622038532
g 0.01872522313619678
h 0.015281114796983544
i 0.09803268275565209
j 2.6009493465114765E-4
k 4.025022852349372E-4
l 0.05713651598535945
m 0.0260223907346016
n 0.07126816163932893
o 0.09471271063524964
p 0.029457363507928328
q 0.007507823000023108
r 0.06604960397321882
s 0.05380042227809845
t 0.06163712565003582
u 0.03394776283426095
v 0.021556173788556679
w 6.206810940538751E-4
x 3.068475365409201E-4
y 7.797474177248249E-4
z 0.006612537543145397
```

5. Advanced Features

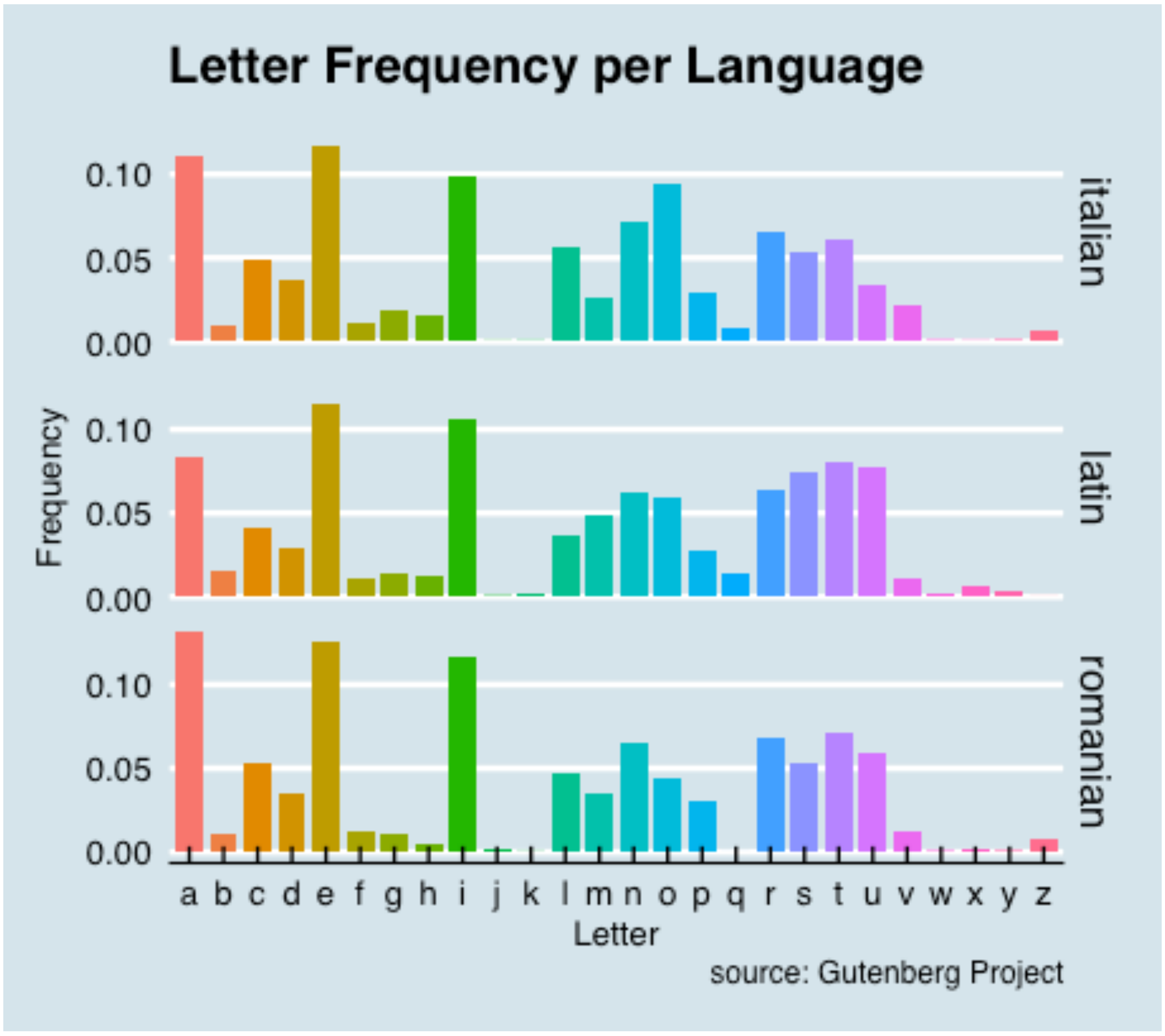
The following advance Features have been used in the code, their usage has been commented in the related section:

- *Counter:* defined in the *Mapper*, counting all the letter mapped, used in the *Reducer* to have a frequency.
- *Combiners:* dedicated FrequencyCombiner Class act as a mini-reducer on the output of the mapper, to have intermediate key-value pairs.
- *Partitioners:* dedicated LetterPartitioner Class redirects *Combiners'* output to one of the two *Reducers*.

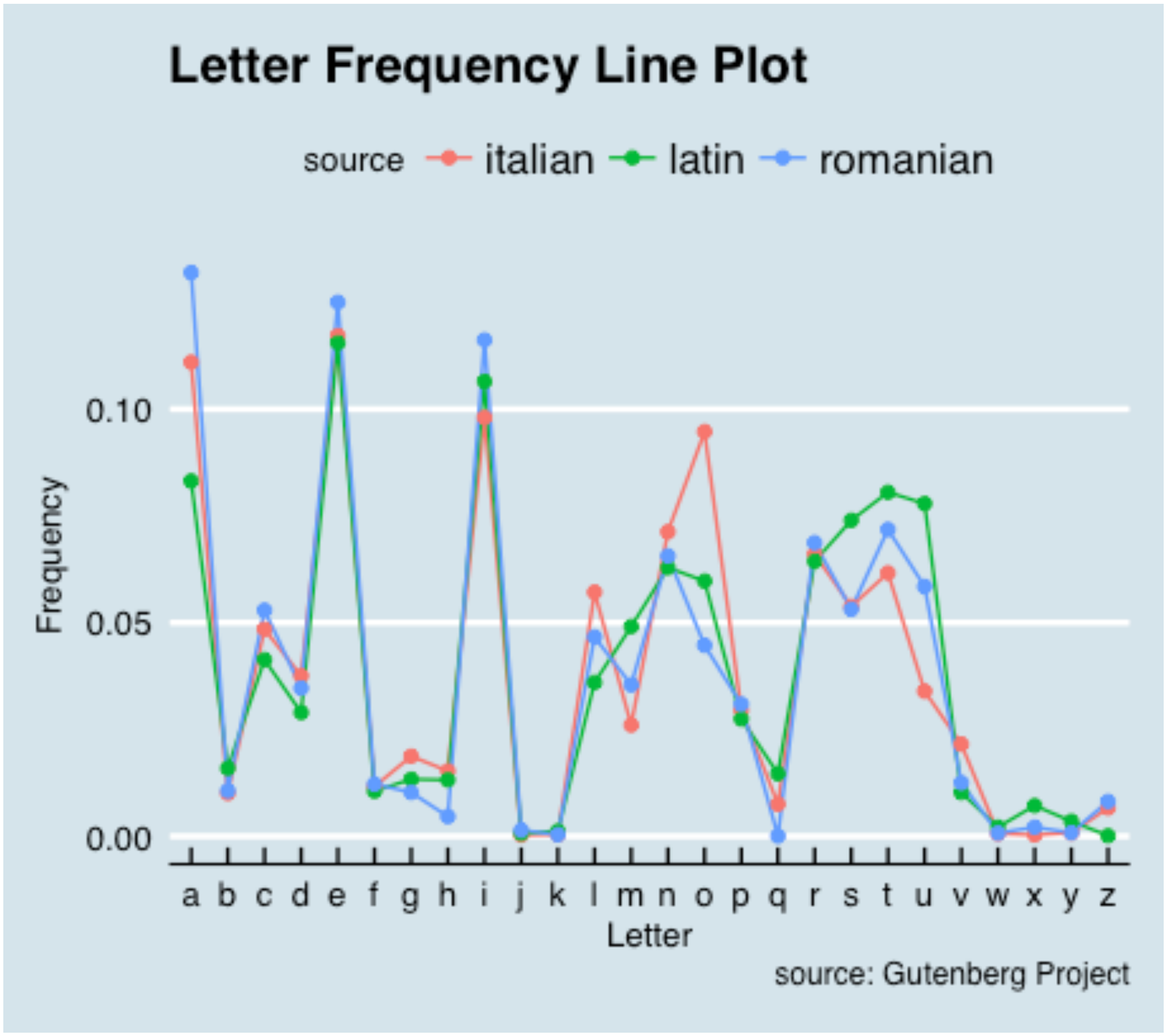
6. Exploratory Data Analysis

The dataset produced by the MapReduce processes were merged back to the host, and analyzed via R in Rstudio. Each dataset was ingested, and, after having their columns properly labeled, they were merged to allow data visualization via *ggplot2* library, Before the merge, an additional column registering the source of the dataset was introduced, to ease overlapping and faceting.

Below a Graph that reports Bar chart for each letter, in each Language. The idioms are stacked to allow comparison, on Y-axis is shown the frequency of occurrence.



The Graph below is showing together Line Charts of the three languages, draw with different colours, to better compare the differences.



The results are showing clearly the roots of Italian and Romanian in Latin; in the three languages, the letter with highest occurrences is a vowel: E for Italian and Latin, A For Romanian. However, A has an occurrence of the same magnitude also in Italian and Latin. I also has frequencies really high, in the three languages, taking it on first places

While O is used a lot in Italian, that letter is passed by some consonant in Latin, and is used way less in Romanian.

Among the consonant, T comes to the first place in Latin and Romanian, while in Italian N and R are the most used. Letter as J, X, W and Y are almost not existing in Romanian and in Italian, while they are present in Latin, sign that something has been lost throughout the centuries.
The letter Q that has some usage in Latin, is used less in Italian, but almost never in Romanian.

In the Snippet presented hereby is available the R code used for the Graphs.

```r
#packages install
install.packages("ggplot2")
install.packages("ggthemes")
# library import
library(ggplot2)
library(ggthemes)

#import italian
italian <- read.table("~/alfa/italianout.txt", sep="\t")
names(italian) <- c("Letter","Frequency")
italian["source"] <- c("italian")
#import latin
latin <- read.table("~/alfa/latinout.txt", sep="\t")
names(latin) <- c("Letter","Frequency")
latin["source"] <- c("latin")
#import romanian
romanian <- read.table("~/alfa/romanianout.txt", sep="\t")
names(romanian) <- c("Letter","Frequency")
romanian["source"] <- c("romanian")

#merging three datasets
output <- rbind(italian, latin,romanian)

# BarChart definition
sp <- ggplot(output, aes(x=Letter, y=Frequency)) +
  geom_bar(stat="identity", width= 0.8,aes(fill=Letter))

# BarChart visualization with faceting
sp + facet_grid(source ~ .) +
  theme_economist() + theme(legend.position="none")+
  labs(title="Letter Frequency per Language",
       caption="source: Gutenberg Project")

# LinePlot definition
p<-ggplot(output, aes(x=Letter, y=Frequency, group=source)) +
  geom_line(aes(color=source))+
  geom_point(aes(color=source))

# LinePlot Visualization
p + theme_economist() +
  labs(title="Letter Frequency Line Plot",
       caption="source: Gutenberg Project")
```