Programming for Big Data – Assignment 1

Hadoop/Map-Reduce

Ciaran Finnegan – TU060 (Part Time) – Second Year
PROG9813: 2022/2023
MSc in Computer Science (Data Science)

Student No : D21124026

26/2/2023

Table of Contents

1 Introduction

1.1 Purpose of Report

This report collates all the delivery requirements for Assignment 1 in the Programming for Big Data module (PROG9813: 2022-23).

The primary content of the report covers;

- How language data (book files) were sourced and then loaded into a HDFS data store on a Linux VM running Hadoop (the VM provided during lecture 1).
- How the Map-Reduce workflow is constructed to ingest the language files stored in HDFS and ultimately compile a breakdown of the average character frequency in each language.
- Design assumptions and decisions made during the implementation of this PBD assignment.
- The Java source written to build the Map-Reduce processes, and how the features offered by Map-Reduce were employed to solve the assignment challenge.
- Python source code and graphs (via a Jupyter Notebook) of the analysis on the output from the Map-Reduce process.

Supporting source code and HDFS outputs accompany this report as part of the overall Brightspace submission.

2 Loading Data Into HDFS

2.1 Setting Up Shared Folders

The VM provided during lectures was used as the Hadoop environment for this assignment. Following lecturer advice, a shared folder on the host machine was set up.

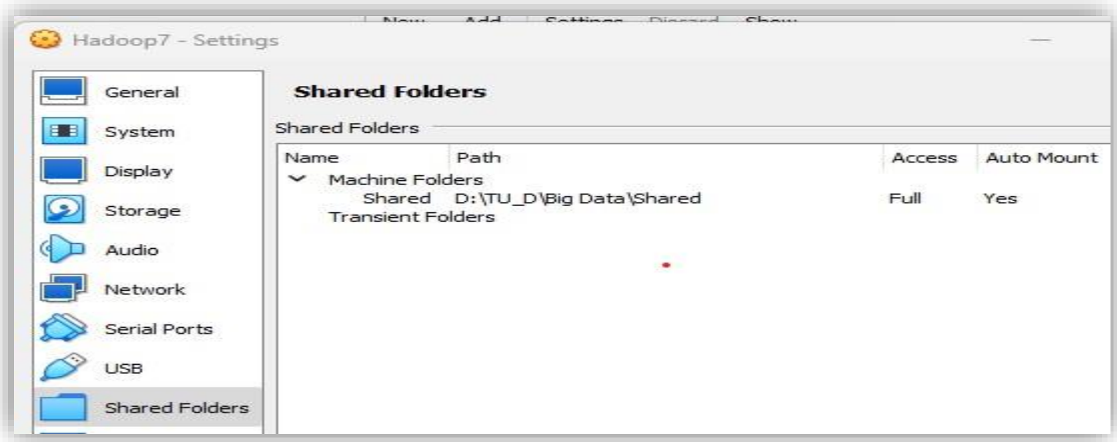


Figure – VM Shared Folder Set Up

2.2 Language Files Source (Books)

Six books were selected from the http://www.gutenberg.org website;

- Two in Spanish.
- Two in German.
- Two in Italian.

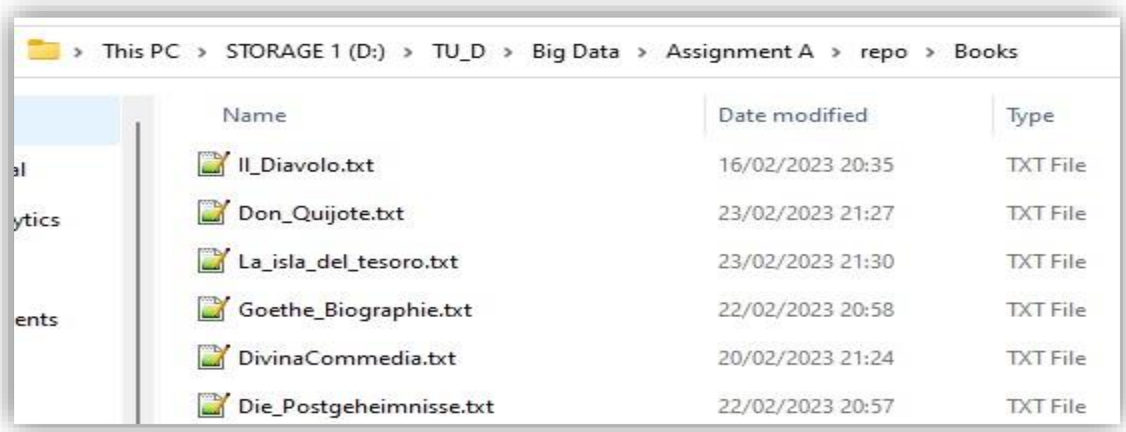


Figure – Book Files Extracted From Gutenberg Site

2.3 Transferring and Loading Data Files

The filenames of each book were altered in the transfer process from host machine to VM to add a suffix indicating language.

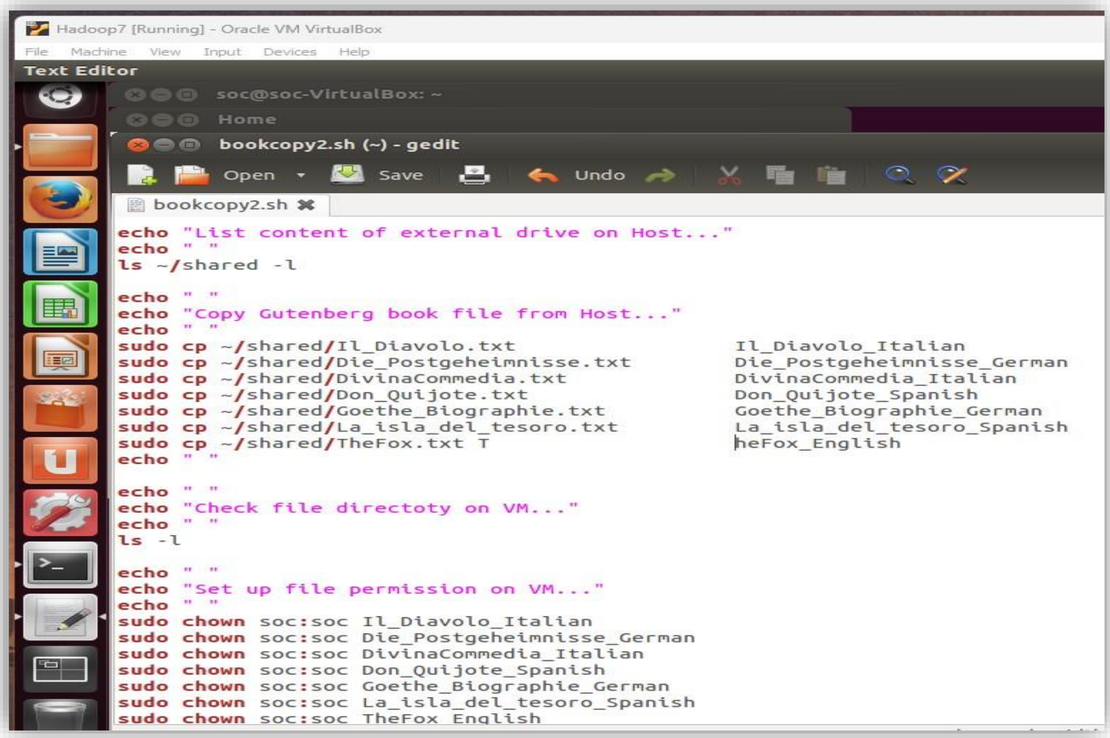


Figure – Shell Script with Linux Commands to Copy/Rename Files in VM

The renaming was done as part of the design assumptions covered in section 3.3 of this report. Setting up a book directory and loading the files into HDFS was done through the following Linux commands;

Figure – Shell Script with Linux Commands to Copy Files to HDFS

The Hadoop interface confirmed that the language book files were successfully loaded.

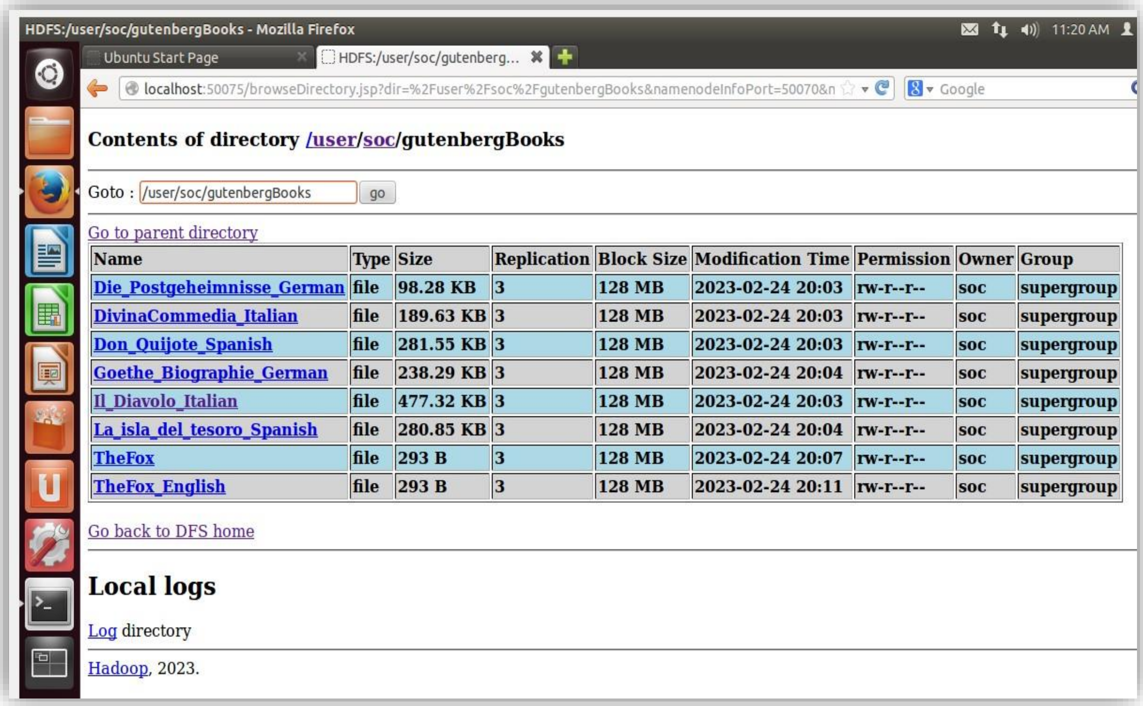


Figure –HDFS Directory Populated with Book Files

A quick inspection (example below) showed that the data was intact and ready for processing.

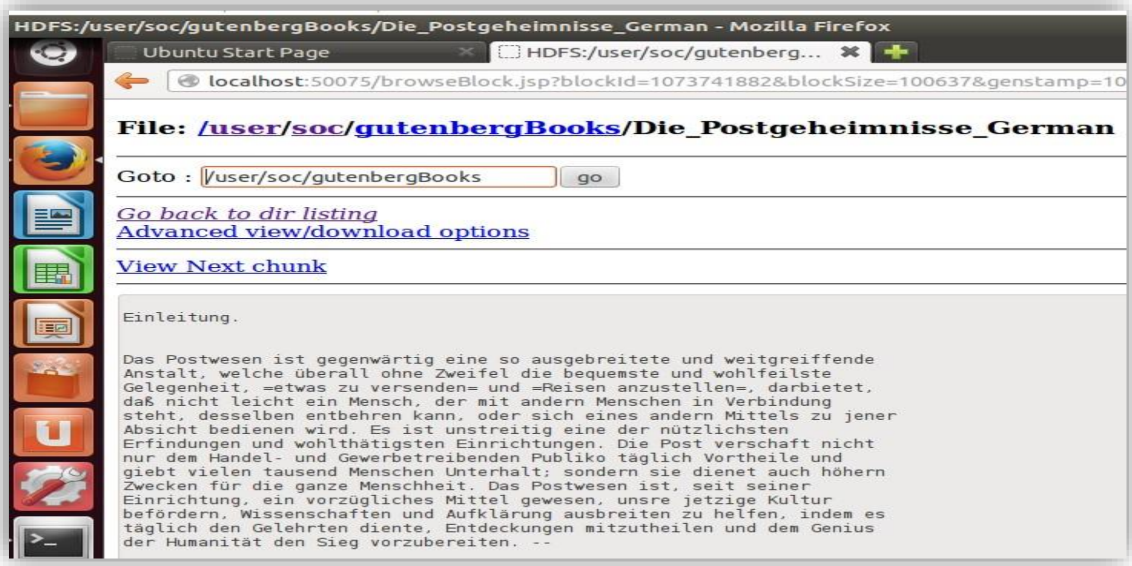


Figure – Example of a German Book File 'Chunk' Stored in HDFS

3 Map-Reduce Process Design

3.1 High Level Overview

This diagram displays the high-level Map-Reduce process built and deployed for the assignment.

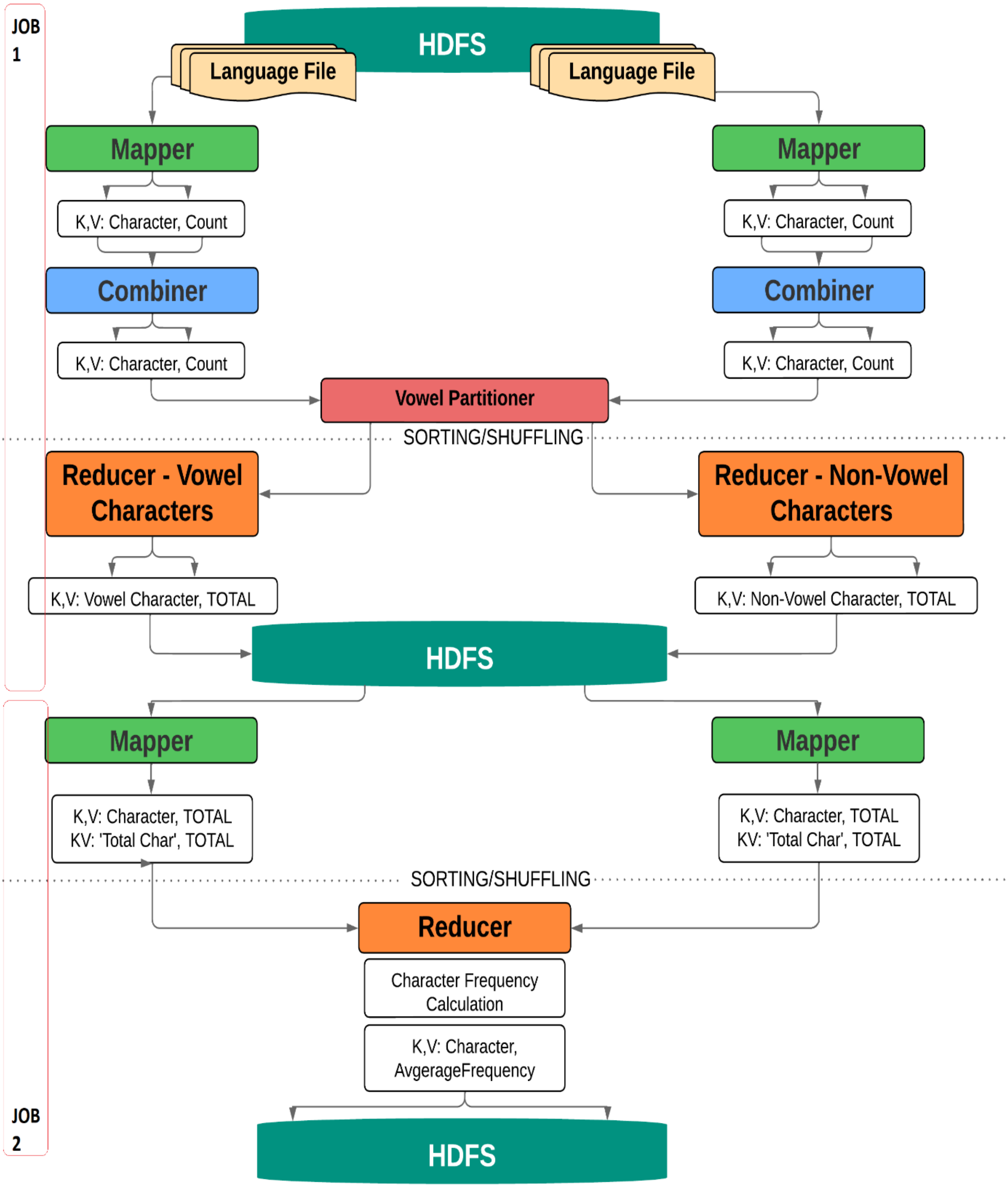


Figure – Map-Reduce Process Overview to Calculate Language Letter Frequency

The following sections of this report elaborate on the individual Map/Reduce processes.

3.2 Explanation of Map-Reduce Workflow

The Java code written for this assignment, compiled in Eclipse under a jar file called `AverageLetterFrequency`, is provided in Section 4 of this report. This section (Section 3.2) provides a generalised overview of the steps in the assignment Map-Reduce process.

3.2.1 Driver Process

There is one Driver process for the entire assignment. Key steps in the process are;

- Invoke jar with command line parameters. This is the main Java project class programme. The `AverageLetterFrequency` jar file is invoked with two parameters.
    - The first is the location of the language files in the HDFS repository.
    - The second is the output directory to be created in HDFS to store the output from the (job1) Map-Reduce process.

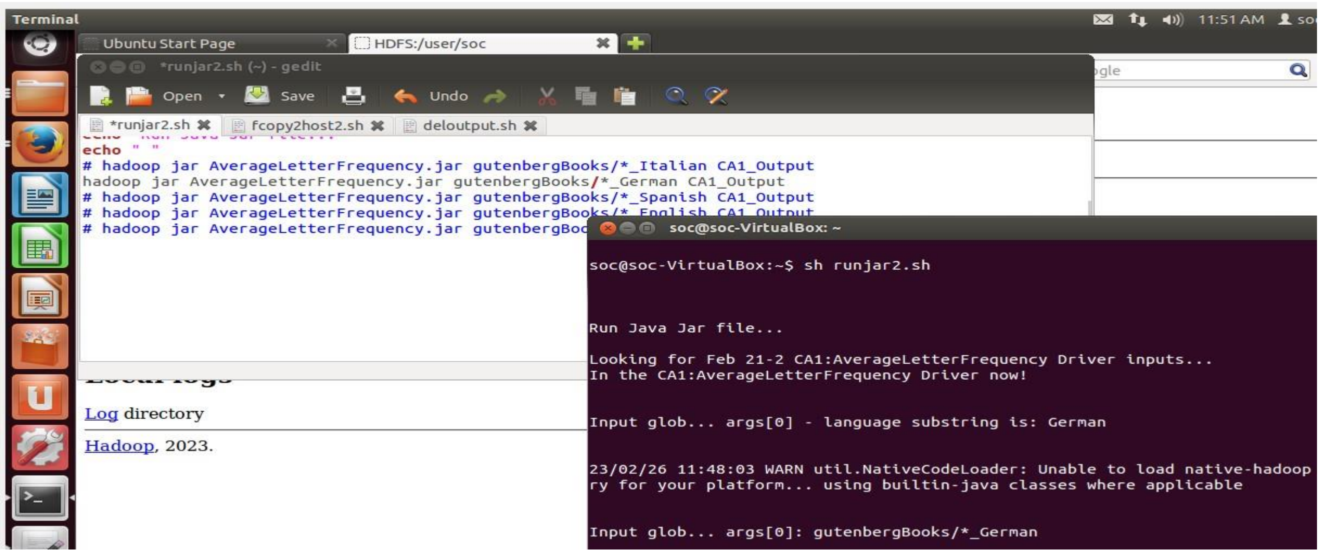


Figure – Cmd Line Parameters to Invoke Jar file for Map-Reduce Assignment

- Use Wildcards to select a subset of language specific book files stored in HDFS (see figure above). This glob is used to set the input path of the first job.
- File name/format is used to indicate language. All files processed by the Map-Reduce process must be in the file name format: ***<bookname>_<language>***. The MR process will read in all files with the given language name used in the command line parameter invoking the jar file (see figure above).
- Mapper and Reducer input and output formats are set up as a Key Value pair of lists of <Text, Integer> datatypes.
- Set up two jobs in the driver class:

- Job 1 reads all the selected language files in the HDFS repository. The Job 1 output to HDFS is a count for the total number of appearances for each individual letter character in the books for the selected languages.

- Job 2 takes the output from Job 1 as an input. Character totals are recreated. The total count of ALL characters in the books is also created in the Job2 MR process. This total of all characters is used to calculate the frequency of each individual language character. The Job 2 Reducer output contains the frequency values for all characters, which is stored in a file in the newly created ***CA1_Output_out*** folder.

- Job1 sets up a Combiner and Partitioner Hadoop process, which are explained in the following sections.
- Job 2 only uses a Mapper and a Reducer process.
- A job sequence is set up so that Job 1 must complete first, before Job 2 is permitted to commence.

3.2.2 Job 1: Mapper/Combiner Process

Key steps in the first (Job1) Mapper process are;

- The function `AverageLetterMapper` is set up to process HDFS input line by line from the specific language files.
- Each line is split into individual characters.
- Non-alpha characters are excluded.
- A regex expression is used to ensure local language characters are included, along with English language characters .
- Counts of characters per line are stored in a Java HashMap and then extracted to write out individual <Char, Count> Key:Value pairs from the Mapper.
- A Combiner class is set up in the Driver class using the same code as the Mapper (`job1.setCombinerClass(AverageLetterReducer.class);`). This acts as a mini-Reducer running locally on the Mapper output and generates the same output format as the Mapper. It is used here to improve Job1 performance.

..

3.2.3 Job 1: Partitioner Process

Key steps in the Partitioner process are;

- A Partitioner class is set up in the Driver code (`job1.setPartitionerClass(AverageLetterPartitioner.class);`).
- The number of Reducer tasks for Job 1 is explicitly set to ‘2’ (`job1.setNumReduceTasks(2);`).

- The choice of Reducer to be selected is determined in the Partitioner process. If a character is a vowel the Reducer processing is directed to Reducer 1. Non vowels are directed to Reducer 2.

- The reason for this partition logic is that there are more occurrences of vowels than non-vowels in a language text. However, the number of vowels is much less (5), therefore this split is used in this MR process distribute effort across the Reducers.

3.2.4 Job 1: Reducer Process

Key steps in the first (Job1) Reducer processes are;

- The function `AverageLetterReducer` is set to read the inputs from the Mapper process, after Hadoop sorting and grouping has taken place.

- Key values, which are individual language characters, are ingested by the Job1 Reducer processes with a list of integer counts for character occurrences in each line of the input.

- The Reducers sum all the character integer counts for each character (Key).

- The output of <*character, Total Count*> is written to HDFS. The two Reducer jobs write to separate files in the ***CA_Output*** directory, which was set in the command line to execute the assignment jar file.

3.2.5 Job 1: MR Process Outputs

Job 1 output is stored under in the ***CA_Output*** directory, created based on the configuration set up in the Driver class.

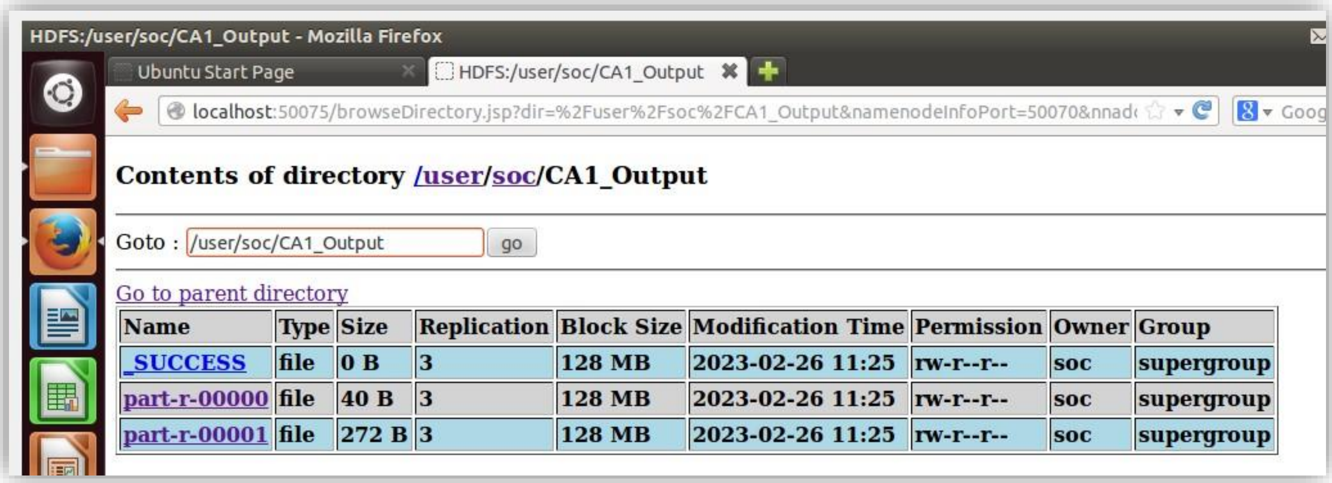


Figure – CA1_Output directory – Job 1 Output

The '***part-r-0000***' file contains the total count of occurrences for vowel characters read in the language file inputs.

Figure – CA1_Output directory – Job 1 Output – Reducer 1 – Vowels

The '***part-r-0001***' file contains the total count of occurrences for non-vowel characters read in the language file inputs (sample represented below).

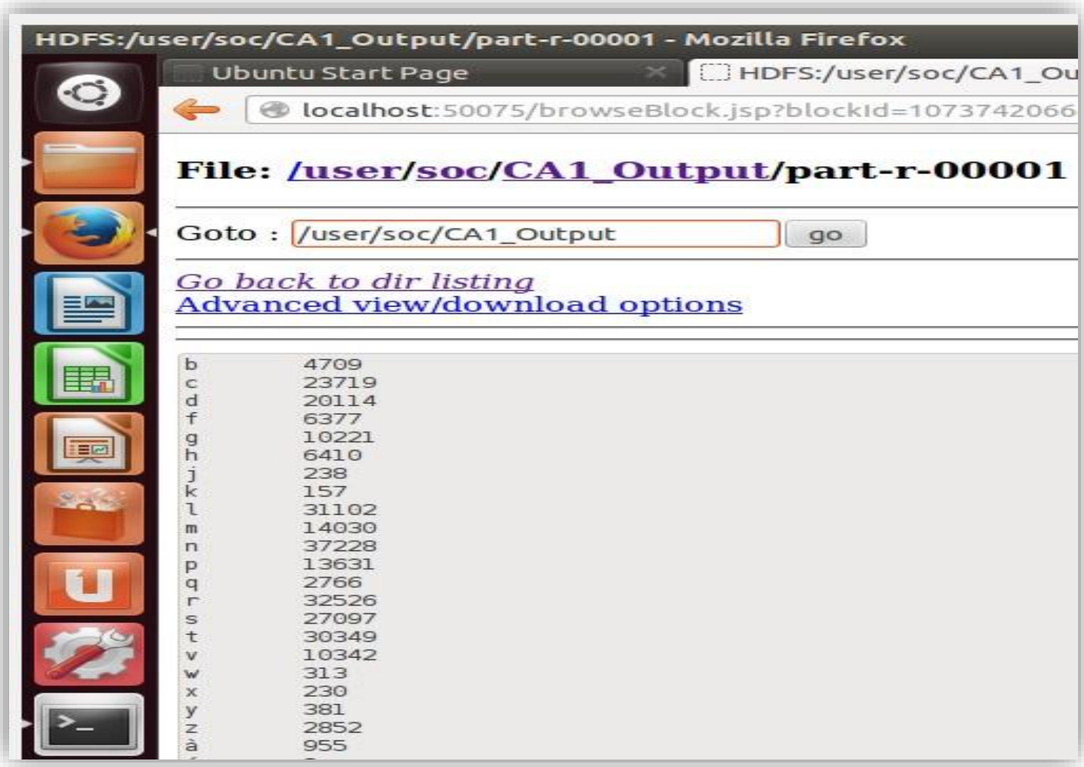


Figure – CA1_Output directory – Job 1 Output – Reducer 2 – Non-Vowels

3.2.6 Job 2: Mapper Process

Key steps in the second (Job2) Mapper process are;

- The `FreqDistrbMapper` class reads the output from Job 1, which is stored in the **CA1_Output** directory in HSFS.

- The input for this Mapper process is much smaller than in Job 1. The Key values in the **CA1_Output** directory are unique, so the Mapper effectively repeats the output process of Job1.

- However, a static key of ***'Total_Chars'*** is also written out by the Mapper as each individual character is re-output with the number of total occurrences. The use of this static key means that it generates a master total output that grows as each separate character is read/output.

- In addition, a '***Local_Char'*** static key is set up to group all non-English language characters. This is done to collate characters such as *ä, ö,* and *ü* (for example) into a single Key category. The purpose of this grouping is to simplify the end result comparison graphs in the Python output for this project.

3.2.7 Job 2: Reducer Process

Key steps in the second (Job2) Reducer process are;

- The *reduce* function in the `FreqDistrbReducer` class performs a simple summation of the values in the iterable Integer list for each individual key. There would be no real sorting or grouping required from Hadoop.

- This summation includes the value for language specific characters such as *ä, ö,* and *ü* (for example), which are grouped under a single Key vale named '***Local_Char'.***

- A static Key called ***'Total_Chars'*** is fed into the Reducer and this summation of the Integer list for this Key will generate the value representing the **total** number of characters in the language book file inputs.

- The *reduce* function does not call the *context.write* routine, as configured by the Driver class. Instead, it incrementally fills a Java HashMap with each Key/Value pair, including the value for the count of all characters.

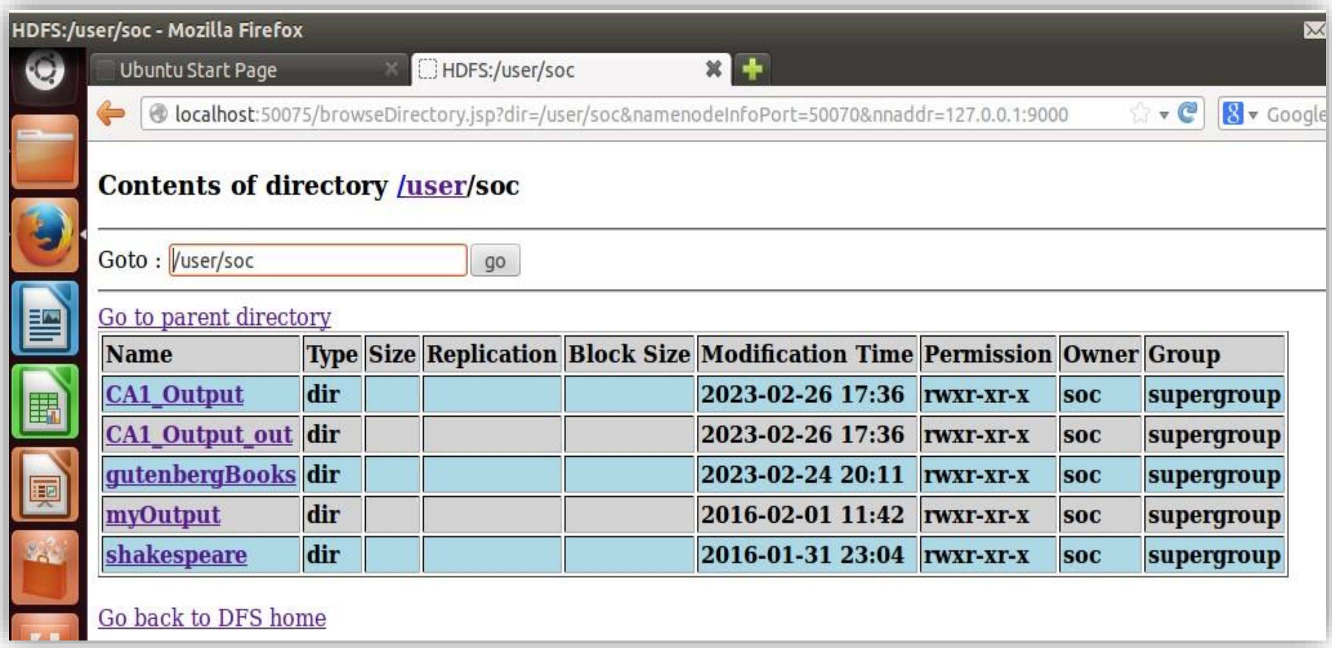
- The Job 2 Reducer has a *cleanup* routine that executes after main processing is complete. This cycles through the Java HashMap created by the *reduce* function and generates a distribution value for each character, based on a division of character totals by total numbers of characters.

- The Key value for final input is concatenated with the name of the language, which has been parsed from the command line input into a Job 2 configuration custom variable called '*language.text*'.

- The *cleanup* routine output is written to a HSFS directory called ***CA1_output_out***.

3.2.8 Job 2: MR Process Outputs

Job 2 output is stored under in the ***CA_Output_out*** directory, created based on the configuration set up in the Driver class.



Contents of directory /user/soc

Goto : /user/soc go

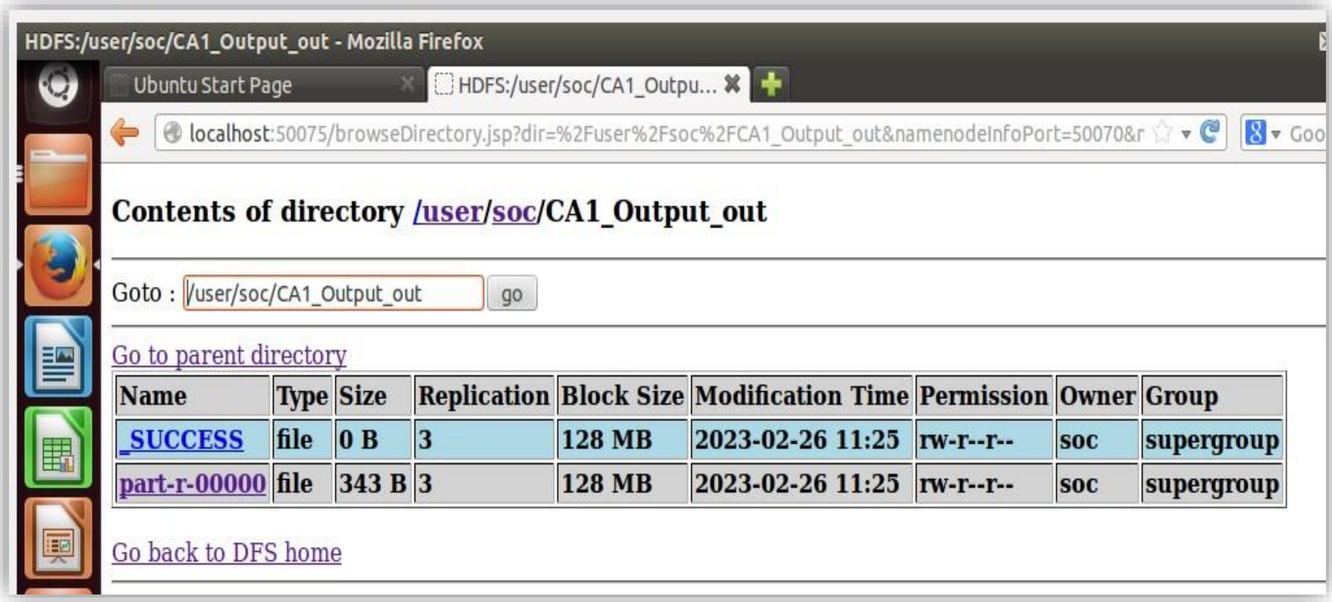
Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
CA1_Output	dir				2023-02-26 17:36	rwxr-xr-x	soc	supergroup
CA1_Output_out	dir				2023-02-26 17:36	rwxr-xr-x	soc	supergroup
gutenbergBooks	dir				2023-02-24 20:11	rwxr-xr-x	soc	supergroup
myOutput	dir				2016-02-01 11:42	rwxr-xr-x	soc	supergroup
shakespeare	dir				2016-01-31 23:04	rwxr-xr-x	soc	supergroup

Go back to DFS home

Figure – CA1_Output_out directory – Job 2 Output

Job 2 output is one file.



Contents of directory /user/soc/CA1_Output_out

Goto : /user/soc/CA1_Output_out go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
_SUCCESS	file	0 B	3	128 MB	2023-02-26 11:25	rw-r--r--	soc	supergroup
part-r-00000	file	343 B	3	128 MB	2023-02-26 11:25	rw-r--r--	soc	supergroup

Go back to DFS home

Figure – CA1_Output_out directory – Job 2 Output File

The ‘***part-r-0000***’ file contains the final breakdown of average letter frequencies and the description of the language (German example represented below).

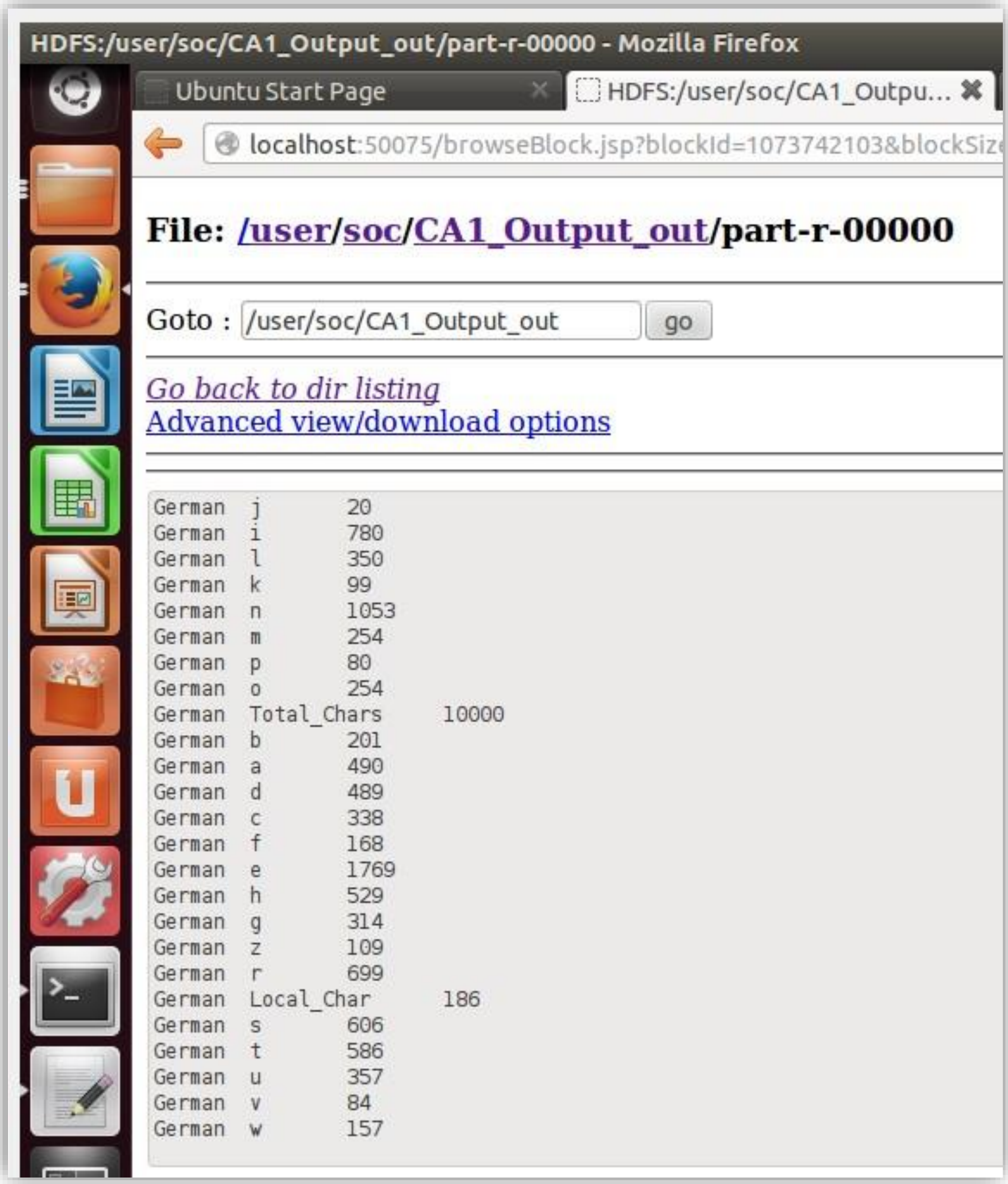


Figure – CA1_Output_out directory – Job 2 Final Output

To maintain the Integer data type for the Job 2 Reducer output, the average letter frequency is multiplied by 10000, The display value is adjusted to a > 1 decimal value during the rendering of the Python bar charts.

3.3 Key Design Decisions and Assumptions

In the design and implementation of PBD CA1 the following were the key design assumptions and decisions;

- Only one jar file is used for the Map-Reduce processing, and this is potentially able to manage multiple files across multiple languages. No hard coding of language names or named file subdirectories in HDFS is used in the Java code. The Java project would not need to be re-compiled to accommodate additional languages (limited to three for this assignment).

- Command line text and file name suffixes are used to alter the selected inputs to the Map-Reduce process and effect the content of the end result formats. All language file names must be in the format: <***bookname_languagename***>, as seen in Section 2.3 of this report. For example, the file *IL_Diavolo_**Italian*** is one of the input books to the MR process in this assignment.

- Globs with a wildcard based on the filename language will pick up all language files for the MR process. In the assignment there are two book files per language but there is no limitation to adding more (and no need to amend source code).

- Two chained jobs are required. One to calculate the total occurrences of each character in the input text, and the second to use the output of the first job to calculate the total number of characters and hence the average letter frequencies (and output).

- The Python code is written in a Jupyter Notebook and also applies the same principle of having no hard coded references to languages. No file name conventions are required. The Python code reads any number of EXCEL files in the local directory of the notebook and calls a graph function. For simplicity, only three graphs are specifically rendered for this assignment, but the code could easily accommodate more language inputs.

- The output from Job2 is downloaded from the HDFS data store after each individual language MR process. The text file is transferred from the VM onto the host machine. The inbuilt EXCEL data ingestion wizard takes the text file and is used to create a three-column output of: ***language – character – frequency count***.

- No data manipulation takes place during the generation of the XL file, and no headers are added.

- The Python graph function takes two parameters; the dataframe with the language letter frequency information loaded from XL; and the name of the language, which is read directly from the dataframe.

4 Map-Reduce: Java Code

4.1 Driver

A description of the operations in this Java code class are given in Section 3.2.1.

AverageLetterFrequency.java | AverageLetterMapper.java | AverageLetterPartitioner.java | AverageLetterReducer.java | FreqDistrbMapper.java

```java

import org.apache.hadoop.conf.Configuration;


// The driver process for the average character frequency MapReduce process
// A single MR process is created and input files on HDFS are read based on
// the command line (glob) input and the naming convention of the book files
// stored on HDFS.


// The book files are all stored with a naming convention of <bookname>_<language>
// For example 'TheFox_English'.
public class AverageLetterFrequency {

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

        // check command line is correctly - two parameters
        if (args.length != 2) {
            System.err.println("Usage: AverageLetterFrequency <input path> <output path>");
            System.exit(-1); }

        // Set Up language from input command line glob
        String cmdInputArg0 = args[0];
        int iUndrScrIndex = cmdInputArg0.indexOf('_');
        String strLang = cmdInputArg0.substring(iUndrScrIndex+1).trim();
        strLang = strLang + '\t';

        // Check Parse of Input globs for language description
        System.out.println("\n\nInput glob... args[0] - language substring is: " + strLang + "\n\n");

        Configuration conf = new Configuration();

        // Set up Job1
        Job job1 = Job.getInstance(conf, "AverageLetterFrequency");

        job1.setJarByClass(AverageLetterFrequency.class);


        // Set Up Mapper and Combiner
        job1.setMapperClass(AverageLetterMapper.class);
        job1.setCombinerClass(AverageLetterReducer.class);
        job1.setMapOutputKeyClass(Text.class);
        job1.setMapOutputValueClass(IntWritable.class);

```

AverageLetterFrequency.java | AverageLetterMapper.java | AverageLetterPartitioner.java | AverageLetterReducer.java | FreqDistrbMapper.java

```java

        // Set Up Partitioner
        job1.setPartitionerClass(AverageLetterPartitioner.class);

        // Set Up Reducer
        job1.setReducerClass(AverageLetterReducer.class);

        // Set number of reducer tasks
        job1.setNumReduceTasks(2);


        // Input and Output format for data
        job1.setOutputKeyClass(Text.class);
        job1.setOutputValueClass(IntWritable.class);

        // Check Input globs
        System.out.println("\n\nInput glob... args[0]: " + args[0] + "\n\n");
        FileInputFormat.addInputPath(job1, new Path(args[0]));
        FileOutputFormat.setOutputPath(job1, new Path(args[1]));



        ControlledJob cJob1 = new ControlledJob(conf);
        cJob1.setJob(job1);


        // Set up Job2
        Job job2 = Job.getInstance(conf, "AverageLetterFrequency Two");

        job2.setJarByClass(AverageLetterFrequency.class);

        // Set Up Mapper - no need for Combiner; data volumes are much lower
        job2.setMapperClass(FreqDistrbMapper.class);
        job2.setReducerClass(FreqDistrbReducer.class);

        // Input and Output format for data
        job2.setOutputKeyClass(Text.class);
        job2.setOutputValueClass(IntWritable.class);

        // Set a custom configuration property that will track the language
        // being analyzed for character frequency distribution in this
        // Map Reduce process
        job2.getConfiguration().set("language.text", strLang);
```

AverageLetterFrequency.java | AverageLetterMapper.java | AverageLetterPartitioner.java | AverageLetterReducer.java | FreqDistrbMapper.java

```
        job2.getConfiguration().set("language.text", strLang);

        ChainMapper.addMapper(job2, FreqDistrbMapper.class, LongWritable.class, Text.class, Text.class, IntWritab
        ChainReducer.setReducer(job2, FreqDistrbReducer.class, Text.class, IntWritable.class, Text.class, IntWrit


        // Job2 reads the files produced on HDFS by Job 1
        FileInputFormat.addInputPath(job2, new Path(args[1]));
        // Job2 output is marked with a suffix
        FileOutputFormat.setOutputPath(job2, new Path(args[1]+ "_out"));

        ControlledJob cJob2 = new ControlledJob(conf);
        cJob2.setJob(job2);


        // Set up Job sequence execution
        // The purpose of the chaining is that Job1 must first generate a count of each character
        // read from the language book files.
        //
        // That data must be in place before Job2 is invoked to generate a count of ALL characters
        // read from the language book files and then calculate the average character distribution
        Job[] jobs = new Job[]{job1, job2};
        for (Job job : jobs) {
          boolean success = job.waitForCompletion(true);

          // Exit process if one of the jobs fails
          if (!success) {
            System.exit(1);
          }
        }

        // Add language description to the final output from the
        // Reducer job after it has completed.

        System.out.println("\nPreparing to end Driver process...\n");

        // No jobs have failed - exit process successfully
        System.exit(0);


        }

}
```

4.2 Job 1

4.2.1 The Mapper

A description of the operations in this Java code class are given in Section 3.2.2.

AverageLetterFrequency.java | AverageLetterMapper.java | AverageLetterPartitioner.java | AverageLetterReducer.java | FreqDistrbMapper.java

```java
import java.io.IOException;

public class AverageLetterMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    private static final Log LOG = LogFactory.getLog(AverageLetterMapper.class);

    static enum LangCharCount{LANG_CHAR_COUNT};

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

        String s = value.toString();
        String strSplit = "";
        String sRegex = "^\\p{L}";
        HashMap<Character, Integer> charCountMap = new HashMap<Character, Integer>();

        System.out.println("In CA1:AverageLetterFrequency Mapper now!");

        // Mapper reads HDFS input line by line
        for (String sInput : s.split(strSplit)) {// Split input into characters
            // Filter out grammar symbols but allow local language characters such as -  ä, ö, ü (for example)
            if (sInput.length() > 0 && sInput.matches(sRegex)) {

                // Convert to lower case - analysis will not differentiate based on case
                String strLCaseWord = sInput.toLowerCase();

                char ch = strLCaseWord.charAt(0);

                // Check that character is valid alpha character - Log output for Hadoop dashboard
                LOG.info("Pre-screening for Character.isLetter() Check..: " + ch);

                if (Character.isLetter(ch)) { // Exclude non-alphabet characters

                        // If the character is already in the map, increment its count...
                        if (charCountMap.containsKey(ch)) {
                            charCountMap.put(ch, charCountMap.get(ch) + 1);
                        }
                        // ...otherwise, add the character to the map with a count of 1
                        else {
                            charCountMap.put(ch, 1);
                        }
                  }

            }
        }

        // Loop through the count of characters in the input read by the Map process
        for (Character ch : charCountMap.keySet()) {

            // Convert character to string for Mapper Output
            String sCharInWord = String.valueOf(ch);
            int ichrCnt = charCountMap.get(ch);

            // Set up data for logging output to first Mapper job
            LOG.info("Mapper output key: " + sCharInWord);
            LOG.info("Mapper output values: " + ichrCnt);

            // Output from Mapper is the count of each character in the word
            context.write(new Text(sCharInWord), new IntWritable(ichrCnt));

        }


    }
}
```

4.2.2 The Partitioner

A description of the operations in this Java code class are given in Section 3.2.3,

AverageLetterFrequency.java | AverageLetterMapper.java | AverageLetterPartitioner.java | AverageLetterReducer.java | FreqDistrbMapper.java

```java

import org.apache.hadoop.io.IntWritable;


public class AverageLetterPartitioner extends Partitioner<Text, IntWritable>{
    private static final Log LOG = LogFactory.getLog(AverageLetterPartitioner.class);
    String partitionkey;

    @Override
    // Input is partitioned into different Reducers based on a check as to
    // whether the character is a vowel or not.
    //
    // Vowels are more commonly used but are a small subset of a Latin alphabet
    // The Reducers are split between processing vowels and non-vowels
    public int getPartition(Text key, IntWritable value, int numPartitions){

        // Set up vowel string
        String vowelCharsToMatch = "aeiou";

        if(numPartitions == 2){
            String partitionKey = key.toString();

            // Compare input key character against vowel string
            if (vowelCharsToMatch.contains(String.valueOf(partitionKey.charAt(0)))){
                LOG.info("numPars: 2 - Partition 0"); // Send Output to Log to indicate Partitioner is working
                return 0; // Partition 0 will be used for vowels
            }
            else {
                LOG.info("numPars: 2 - Partition 1"); // Send Output to Log to indicate Partitioner is working
                return 1; // Partition 1 will be used for non-vowels
            }
        } else if (numPartitions == 1) { // Default
            return 0;
        }
        else {
            System.err.println("AverageLetterPartitioner can only handle 1 or 2 partitions");
            return 0;
        }
    }
}

```

4.2.3 The Reducer

A description of the operations in this Java code class are given in Section 3.2.4.

AverageLetterFrequency.java | AverageLetterMapper.java | AverageLetterPartitioner.java | AverageLetterReducer.java

```java

import java.io.IOException;


public class AverageLetterReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    private static final Log LOG = LogFactory.getLog(AverageLetterReducer.class);

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {

        List<IntWritable> valueList = new ArrayList<IntWritable>();


        // This code block takes the Iterable list input and reads into
        // a new list
        for (IntWritable val : values) {
            valueList.add(new IntWritable(val.get()));
        }

        // The purpose of this code is to read the key/value inputs into
        // new variables and use the LOG function to ouput to the Hadoop
        // dashboard to show that the Mapper input to the first Reducer
        // process is correct.
        StringBuilder sb = new StringBuilder();
        for (IntWritable v : valueList) {
            sb.append(v.get());
            sb.append(", ");
        }
        if (sb.length() > 0) {
            sb.delete(sb.length() - 2, sb.length());
        }

        // Output to log file in Hadoop dashboard
        String valuesAsString = sb.toString();
        LOG.info("Reducer input key: " + key);
        LOG.info("Reducer input values: " + valuesAsString);


        // Initialize counter variable
        int iChrCount = 0;


        // Count the total number of characters in the book
        // The use of the Iterable list in the LOG output requires
        // the use of the 'new' list to provide Reducer function
        // output.
        //
        // (Once the Mapper Iterable list is processed for the LOG
        // it cannot be reset).
        for (IntWritable vi : valueList) {

            // Read the Mapper/Combiner output and increment counter
            iChrCount += vi.get();

        }


        // Write Reducer Output - ignore very infrequent characters
        // If, after reading entire books in the language, the character
        // count is less than 5 then the character can be considered spurious
        if (iChrCount > 5) {
            context.write(key, new IntWritable(iChrCount));
        }
    }

}
```

4.3 Job 2

4.3.1 The Mapper

A description of the operations in this Java code class are given in Section 3.2.6.

AverageLetterFrequency.java | AverageLetterMapper.java | AverageLetterPartitioner.java | AverageLetterReducer.java | FreqDistrbMapper.java

```java
import java.io.IOException;

// This is the second (chained) Mapper process
// This reads the output from the first MapReduce Job, which is a file with
// a list of each character found in the language book files on HDFS, along
// with the count of the occurrence of those characters.
public class FreqDistrbMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    private static final Log LOG = LogFactory.getLog(FreqDistrbMapper.class);

    // The purpose of this Map process is to create a Key/Value pair that uses a static key text to
    // record a value that represents the total count of ALL characters and which will be used for
    // average frequency distribution calculations in the Job 2 Reducer function.
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

        String s = value.toString();
        String strSplit = "\\W+";
        String strTotalLabel = "Total_Chars"; // Static Key Value set to record ALL characters

        // Each line in the Job 1 output is read.
        // Each line represents a character and character count
        String[] parts = s.split(strSplit);

        // The first line input represents the individual language character
        String sCharCapture = parts[0].trim();
        // The second line represents the count of the character
        int iCharCntInt = Integer.parseInt(parts[1].trim());

        // 'Local' (non-English) characters will not be read - but their count is
        // The blank character is interpreted as one of these non-English characters
        // (such as ä, ö, ü) and they are grouped under a default key value
        if (sCharCapture.isEmpty()){
            sCharCapture = "Local_Char";
            LOG.info("Checking for empty key-value: " + sCharCapture + "-" + iCharCntInt);
        }

        // This line rewrites the character and count to preserve this data for the Job 2 Reducer
        context.write(new Text(sCharCapture), new IntWritable(iCharCntInt));
        // This line writes out a static key value to ensure the Reducer is fed a count of
        // total characters read from the language book files.
        context.write(new Text(strTotalLabel), new IntWritable(iCharCntInt));

    }
}
```

4.3.2 The Reducer

A description of the operations in this Java code class are given in Section 3.2.7,

AverageLetterFrequency.java | AverageLetterMapper.java | AverageLetterPartitioner.java | FreqDistrbReducer.java | A

```java

import java.io.IOException;



// This is the second (chained) Reducer job
// The Job 2 Mapper function uses a static key value to allow for a count of all
// characters. This allows the Job 2 Reducer function here to calculate the
// average frequent for each character and generate the final output
public class FreqDistrbReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
        private static final Log LOG = LogFactory.getLog(FreqDistrbReducer.class);
        private HashMap<Text, Integer> totals = new HashMap<Text, Integer>();
        private String strTotalLabel = "Total_Chars";
        private Text languageText;

        // The Driver function sets up a language text output for the final HSFS Reducer
        // output based on the command line (glob) used to run the MR jar file
        protected void setup(Context context) throws IOException, InterruptedException {
            // Read the language description used as the parameter to run the jar file
            // from the command line.
            languageText = new Text(context.getConfiguration().get("language.text"));
        }

        public void reduce(Text key, Iterable<IntWritable> values, Context context)
                throws IOException, InterruptedException {

            // This is similar to the reducer in Job 1 as a total is counted
            // of all occurrences of the Key values

            // For Job2, one of the inputs reflects the count of ALL characters,
            // which has been recorded against a static 'Total_Chars' key.
            int sum = 0;

            for (IntWritable value : values) {
                sum += value.get();
            }

            if (totals.containsKey(key)) {
                sum += totals.get(key);
            }

            totals.put(new Text(key.toString()), sum);
        }

```

AverageLetterFrequency.java | AverageLetterMapper.java | AverageLetterPartitioner.java | FreqDistrbReducer.java

```java
        // This function is run to process the end result Reducer data and perform the
        // calculations to generate the frequency distribution of characters
        protected void cleanup(Context context) throws IOException, InterruptedException {
            float iTotalCharsCnt = 0;

            // Create a HashMap with the Reducer dataset
            for (Map.Entry<Text, Integer> entry : totals.entrySet()) {
                String sChkTotal =  entry.getKey().toString().trim();

                // Store the value that represents the total count of ALL
                // characters read by the MapReduce process
                if (sChkTotal.equals(strTotalLabel)){
                    iTotalCharsCnt = entry.getValue();
                }
            }

            // Use the count of ALL characters to loop through the count for individual
            // characters and produce a frequency distribution value for each one and
            // output the result as the final Reducer result to HDFS.
            for (Map.Entry<Text, Integer> entry2 : totals.entrySet()) {
                float iEnt = (entry2.getValue());
                float iCalField = ((iEnt) / iTotalCharsCnt);
                int iDistrb = (int) (iCalField * 10000);

                // Output LOG data for the Hadoop dashboard logs on the second (Job2)
                // reducer output.
                LOG.info("Working through second for-loop - Key-Value-Calc-Int: "
                                                    + entry2.getKey()+ "-"
                                                    + entry2.getValue() + "-"
                                                    + iCalField + "-" + iDistrb);

                // Datatype manipulation to format output of Reducer job
                String sTxt1 = languageText.toString();
                String sTxt2 = entry2.getKey().toString().trim();
                String sTxt3 = sTxt1+sTxt2;
                Text keyText = new Text(sTxt3);

                // If character is very infrequent it can be ignored as a spurious
                // inclusion in the language documents
                if (iDistrb > 10) { // Appears less that 0.1%
                    //Generate final Job 2 Reducer outputs
                    context.write(keyText, new IntWritable(iDistrb));
                }
            }
        }
}
```

5 Python: Graph Analysis of MR Outputs

5.1 Set Up Excel From HSFs Output

The final output from the Map-Reducer process is downloaded on the VM and then copied to the host machine.

The text file is loaded into Excel and the inbuilt wizard creates an *.xlsx file output wit the data in three columns (no headers).

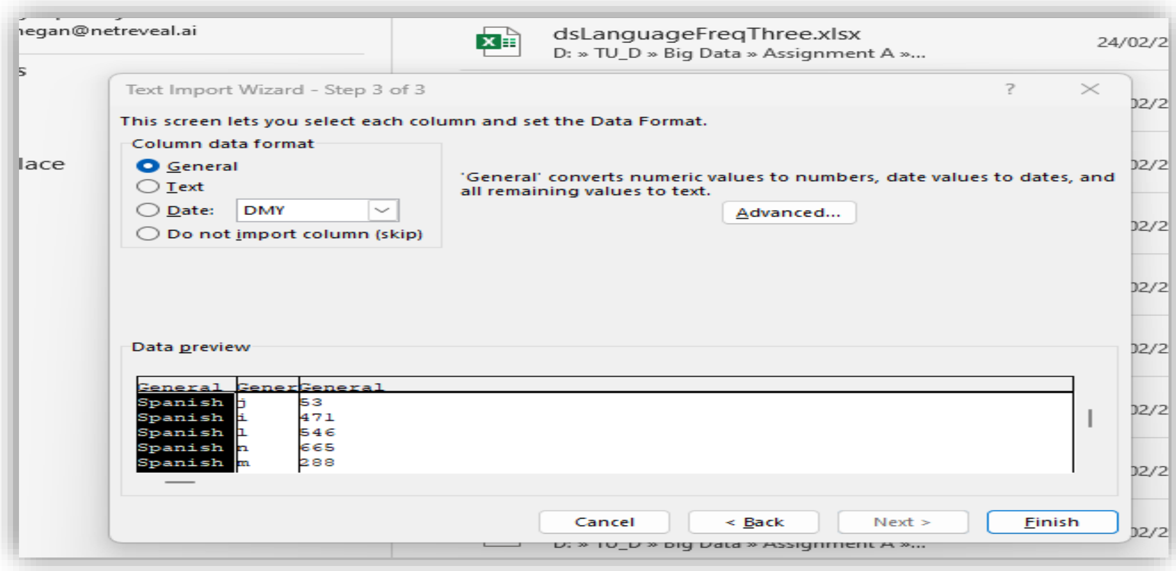


Figure – Loading HDFS Output Into Excel

All *.xlsx files are then copied to the same local directory from which the Python Jupyter Notebook written for the assignment is stored.

5.2 Python Jupyter Notebook

The Python code reads all locally stored *.xlsx files (all of which are assumed to be in the HDFS output format) and loads this language data into separate Panda dataframes.

The notebook generates a character frequency graph for each language dataframe.

Below is the Jupyter Notebook code with generated output included;

Programming for Big Data - Continuous Assessment 1 - Class Group: TU060

MSc in Computer Science - Data Science (Part Time - Yr 2)

Student Name: Ciaran Finnegan

Student No: D21124026

Student EMail: D21124026@mytudublin.ie

February 2023

This Python expects all MapReduce output files to be converted to EXCEL and stored in the local programme directory.

Language names are read from the file content (not hard coded in the file name).

The Python code is written to read any number of input language files, but for simplicity only three language graphs are created.

Table of Contents

1 Import Python Libraries

1.1 Import Libraries

Note: It is assumed that the matplotlib and plotly library has been installed on the user's machine before it is available for import.

> pip install plotly

1.1.1 Panda + Numpy

In [1]:
```
# Python libraries for data manipulation - dataframes
import pandas as pd
```

1.1.2 Plotly Libraries

In [2]:
```
# For interactive graphics
import plotly.express as px
```

1.1.3 Library for file loading using Wildcards

In [3]:
```
import glob
```

2 Language Data Analysis

What are the letter frequencies across languages?

2.1 Set Up Graph Function

Single code block - function is called multiple times to display a language graph.

In [4]:
```
## This is a function to generarte the character distibution graph.

## All language outputs from the MR processes are identical in format.
## Hence a single function can be written to generate all graphs.

def generate_CharFreqGrpah(df_lang, language): ## Language read from XL file contents directly

    # Sort the dataframe so that the x-axis is in alphabetical order
    df_lang = df_lang.sort_values('Character')

    # Bar Chart on Letter frequency
    figCharFreq = px.bar(df_lang,
                         x='Character',
                         y=df_lang['Frequency']/10000, # Adjust format of Y Axis
                         template="simple_white"  # Use a clearer template
                         )

    figCharFreq.update_yaxes( # Set up the y-axis format
        title=dict(
            font_size=20,
            text="Letter Frequency",
            ),
        tickformat='.2f',
        showgrid=True
    )

    figCharFreq.update_xaxes( # Set up the x-axis format
        title=dict(
            font_size=20
            ),
        ticks="outside",
        showgrid=True
    )

    figCharFreq.update_layout( # Customize font and legend orientation & position
        font_family="Rockwell",
        height=700,
        title=dict( # Dynamic graph title
            text="Average Distribution of Letter Frequency for the " + language + " language",
            font_size=20
            ),
        margin=dict(
              l=10,
              r=60,
              b=10,
              t=150
            )
         )

    # Display Bar Chart
    figCharFreq.show()

    return 1
```

2.2 Ingest Language Files into Panda Dataframes

2.2.1 Read Language Data Ouputs from MapReducer Processes

In [5]:
```
# Set up custom colum names for dataframe
names=['Language','Character', 'Frequency']
```

In [6]:
```
# Define the wildcard  cpattern to search for all Excel files in local directory
xlLanguage_files = glob.glob('*.xlsx')
```

In [7]:
```
# Set up list of language dataframes
dfs_lanuagelist = []
```

In [8]:
```
# Loop through the EXCEL files ouput from the MAPReducer processes
# and read each one into a dataframe list
for xl_file in xlLanguage_files:
    df = pd.read_excel(xl_file, header=None, names=names)
    dfs_lanuagelist.append(df)
```

2.3 Display Language Dataframes

2.3.1 Display Character Frequency Dataframes

The output from the dataframe list shows the format of the data after being read from the EXCEL file source.

The dataframe has the following structure:

```
> Each line has the name of the 'Language'. The upper most value in the dataframe is read by the graph function for
dynamic title display.

> The 'Total Character' value used in the calaculations second (chained) MapReduce process in Hadoop is removed to avoid
distorting the graph.

> The dataframe is put in an English language centric order (a,b,c, etc.). This is done to make visual comparisons
across langauges graphs more straightforward.

> The 'Freqency' represents an average distribution number of the character. It is calculated by the division of
the number of occurences of an inidividual charcter recorded in the MR process by the total number of characters
read in from the langauge book inputs. The value has been scaled up into an larger Integer value in the MR
process (to simplify the last Reduce process output) but is adjusted during graph generation.
```

In [9]:
```python
# Loop through language dataframe for some basic data clean up and sorting

for df_language in dfs_lanuagelist:
    # Remove the Total Chars row from dataframes - this was used in the MapReduce processing but is not required for grpah
    df_language.drop(df_language.index[(df_language["Character"] == "Total_Chars")],axis=0,inplace=True)

    # Sort the dataframe so that the x-axis is in alphabetical order
    df_language = df_language.sort_values('Character')

    # Display dataframe for each language - break space included to seperate language files
    print(df_language)
    print("\n")
    print("\n")
```

```
   Language  Character  Frequency
20  Spanish  Local_Char        298
9   Spanish          a       1176
8   Spanish          b        173
11  Spanish          c        372
10  Spanish          d        507
13  Spanish          e       1322
12  Spanish          f         47
15  Spanish          g        102
14  Spanish          h        112
1   Spanish          i        471
0   Spanish          j         53
2   Spanish          l        546
4   Spanish          m        288
3   Spanish          n        665
6   Spanish          o        926
5   Spanish          p        231
18  Spanish          q        163
19  Spanish          r        636
21  Spanish          s        723
22  Spanish          t        412
23  Spanish          u        456
24  Spanish          v        118
16  Spanish          y        143
17  Spanish          z         38
```

```
   Language  Character  Frequency
18  Italian  Local_Char        101
8   Italian          a       1063
7   Italian          b         91
10  Italian          c        461
9   Italian          d        391
12  Italian          e       1153
11  Italian          f        124
14  Italian          g        198
13  Italian          h        124
0   Italian          i       1061
1   Italian          l        604
3   Italian          m        272
2   Italian          n        724
5   Italian          o        960
4   Italian          p        265
16  Italian          q         53
17  Italian          r        632
19  Italian          s        527
20  Italian          t        590
21  Italian          u        314
22  Italian          v        201
15  Italian          z         55
```

```
   Language  Character  Frequency
19   German  Local_Char        186
10   German          a        490
9    German          b        201
12   German          c        338
11   German          d        489
14   German          e       1769
13   German          f        168
16   German          g        314
15   German          h        529
1    German          i        780
0    German          j         20
3    German          k         99
2    German          l        350
5    German          m        254
4    German          n       1053
7    German          o        254
6    German          p         80
18   German          r        699
20   German          s        606
21   German          t        586
22   German          u        357
23   German          v         84
24   German          w        157
17   German          z        109
```

2.4 Display Language Graphs

This Python code will read any number of language file inputs.

However for the purpose of this assignment only the three following langauge files are displayed:

> Spanish.

> Italian.

> German.

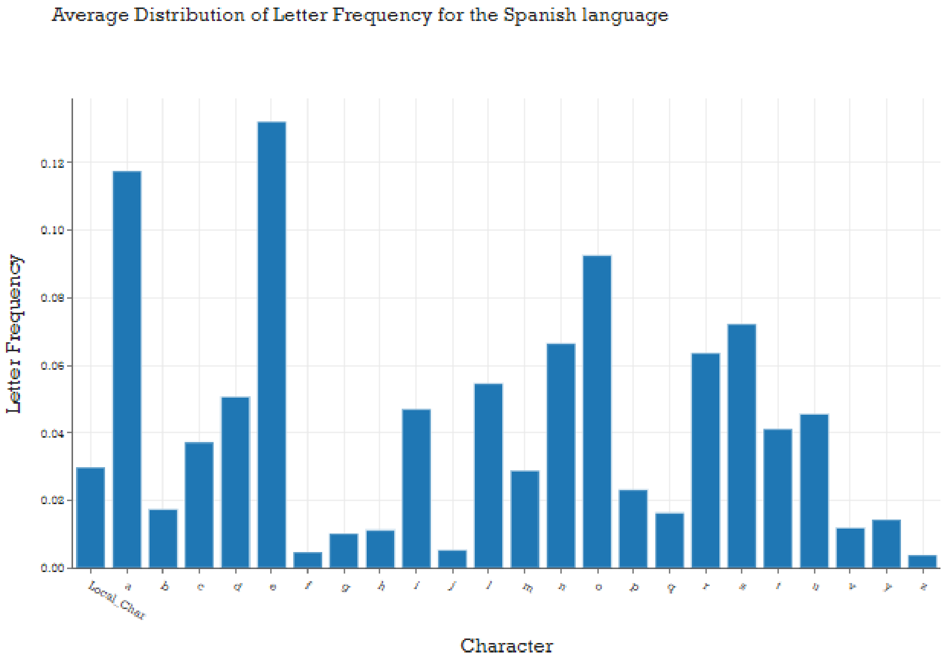
The reference to 'Local Character' reflects the decision to group non-English characters such as ä, ö, ü in a single category.

These local language characters are grouped under a single heading in the MR process and the average distribution calculated accordingly.

This decision does remove certain data granularity but makes cross language visual comparisons of the language graphs more straightforward.

2.4.1 Display Bar Chart Graph For Spanish

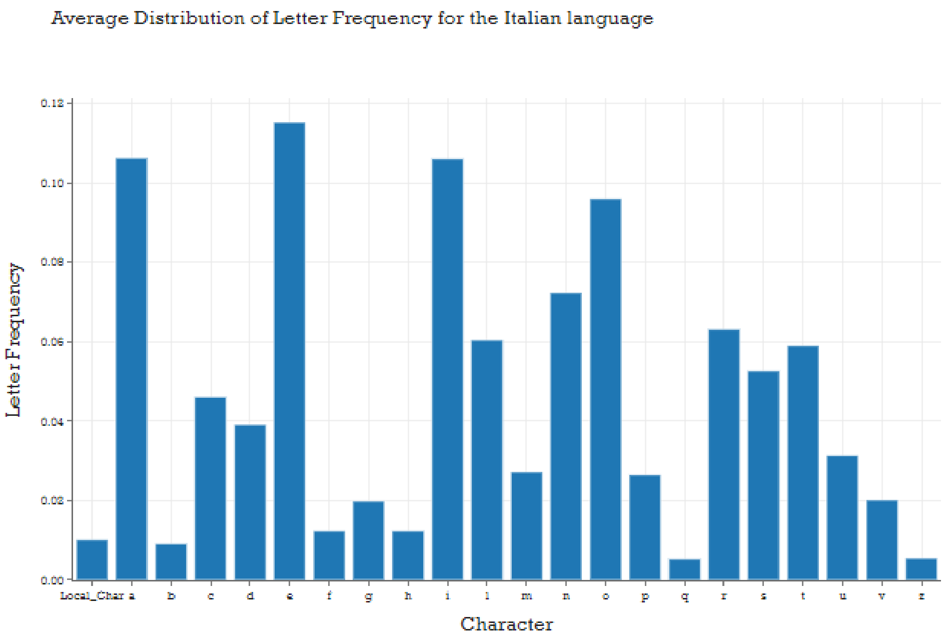
In [10]:
```
# First Language Dataframe - Invoke Graph generation function with Language dataframe
result = generate_CharFreqGrpah(dfs_lanuagelist[0], dfs_lanuagelist[0].iloc[0,0])
```



The frequency of the 'a', 'e', and 'o' characters in Spanish is not unlike English. (An English language character distribution graph is included in the brief for this assignment).

2.4.2 Display Bar Chart Graph For Italian

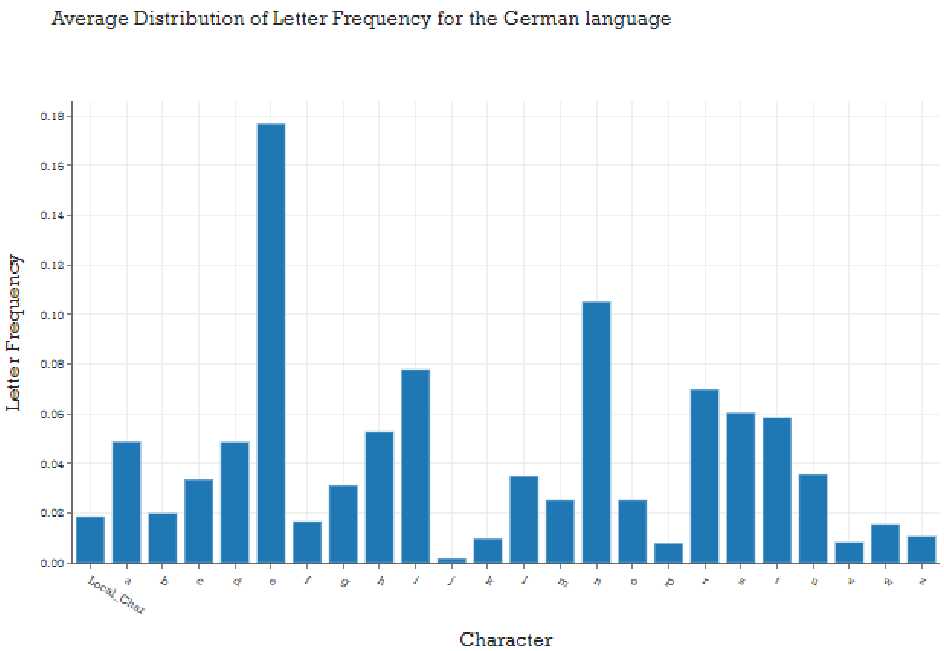
In [11]:
```
# Second language Dataframe - Invoke Graph generation function with language dataframe
result = generate_CharFreqGrpah(dfs_lanuagelist[1], dfs_lanuagelist[1].iloc[0,0])
```



The frequency of 'i', and 'o' characters in Italian is more common than in English.

2.4.3 Display Bar Chart Graph For German

In [12]:
```
# Third language Dataframe - Invoke Graph generation function with language dataframe
result = generate_CharFreqGrpah(dfs_lanuagelist[2], dfs_lanuagelist[2].iloc[0,0])
```



The frequency of 'e', and 'n' characters in German is significantly more common than in English.

<image>

5.3 Analysis Commentary

Commentary on each language graph is included in the output in Section 2.4 of the Python Jupyter Notebook.

A brief description is provided on how the distribution of characters for each language varies from English. The average letter frequency for English given in the brief for PBD Assignment 1 is used as the basis for this comparison.

Given that the assignment brief shows a letter frequency based on the Oxford English dictionary, it would be necessary to greatly increase the size of input for ‘foreign’ languages in our VM to give a more accurate comparison. Resource limitation for this CA meant that it was necessary to restrict the size of the Map-Reduce input.