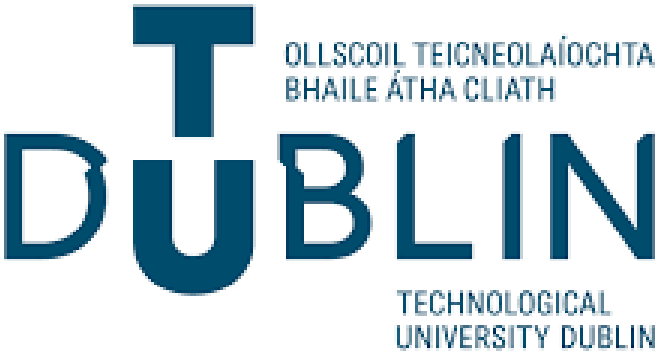
TRACKING LETTER FREQUENCIES ACROSS VARIOUS LANGUAGES USING HADOOP

Sean Kelly (D18124684) – Year 2, MSc Advanced Software Development (ASD) (TU060)

Programming for Big Data (PROG9813) – Brendan Tierney



Programming for Big Data 2021

Table of Contents

Explanation of the steps you performed for loading the data sets into HDFS.

Starting and terminating the Hadoop Service

To make things quicker I created a shell script to automate the starting and finishing commands, so they do not have to be entered one by one each time I start up or close down the virtual machine.

Name of starting shell script: start.sh

Content in starting shell script:

```
start-dfs.sh
start-yarn.sh
mr-jobhistory-daemon.sh start historyserver.
```

To run the shell script, navigate to directory where it is located and enter ./start.sh to run it. This will trigger the three commands contained in it and will start up the Hadoop service so MapReduce functions can be run on the virtual machine.

Like above I created a shell script named finish.sh to close the Hadoop services before powering off the virtual machine.

```
stop-dfs.sh
stop-yarn.sh
mr-jobhistory-daemon.sh stop historyserver
```

Loading data sets into HDFS

For this assignment I downloaded two books in English, French and German and saved them to the directory /home/soc/Desktop/books on the virtual machine. These books will be used as the data set for analysing the frequencies of letters across the three languages. To load them into HDFS the syntax for the terminal command is: hadoop fs -put FILEPATH-IN-YOUR-LOCAL-DIRECTORY FILEPATH-ON HDFS. It is important to make sure that the filepath on HDFS does not already exist or else the data will no be loaded. The syntax for the command is: hadoop fs -put | FILEPATH-IN-YOUR-LOCAL-DIRECTORY | NEW-FILEPATH-ON HDFS

```
hadoop fs -put /home/soc/Desktop/books /user/soc/books
```

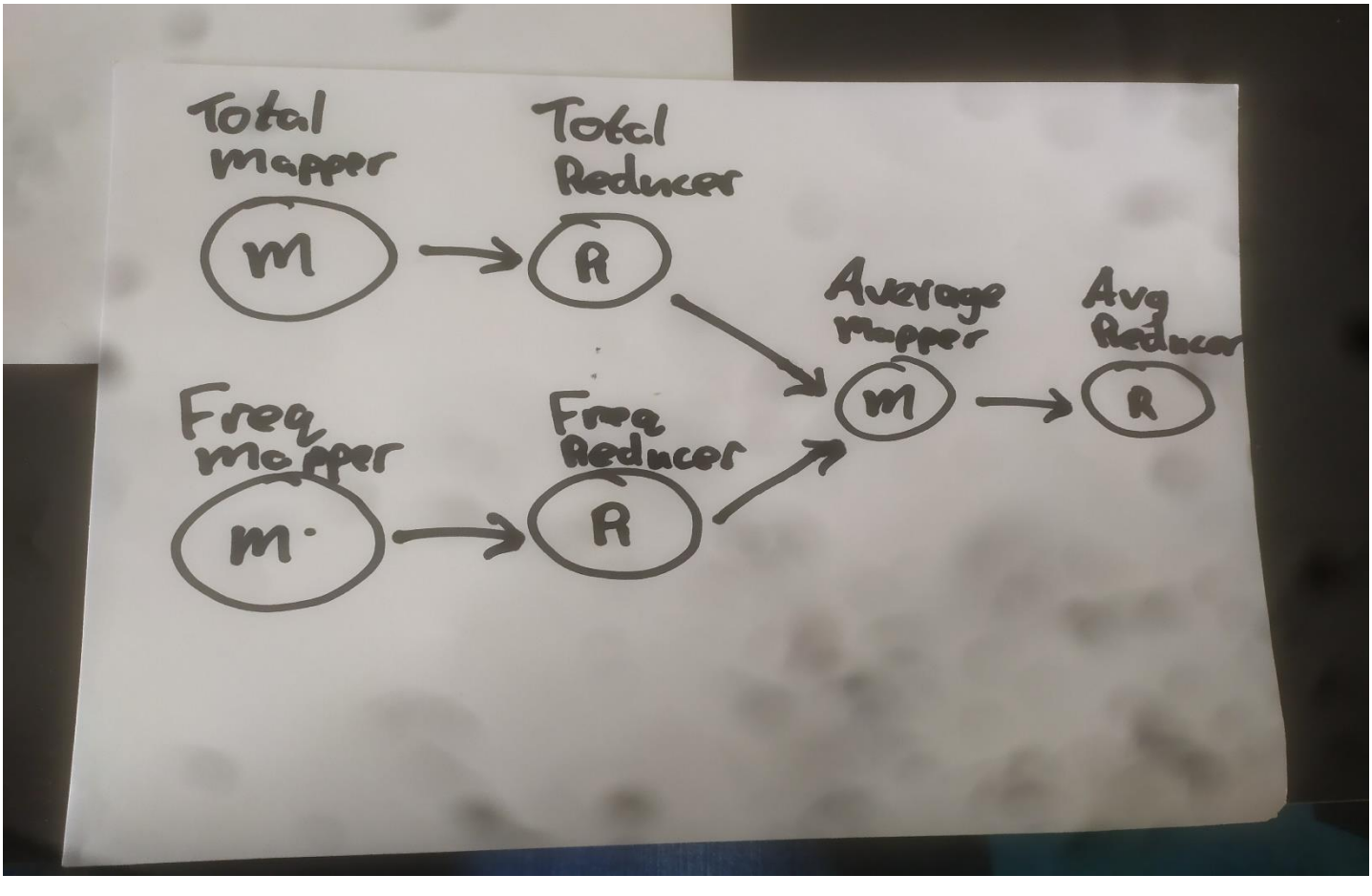
Detailed design, including diagrams and detailed explanations of each part of the Map-Reduce process.

Design overview:

This mapreduce process will involve chaining multiples jobs in order to produce the desired output of the average occurrences of each letter per language. The data set used included two books from three languages: English, French and Polish.

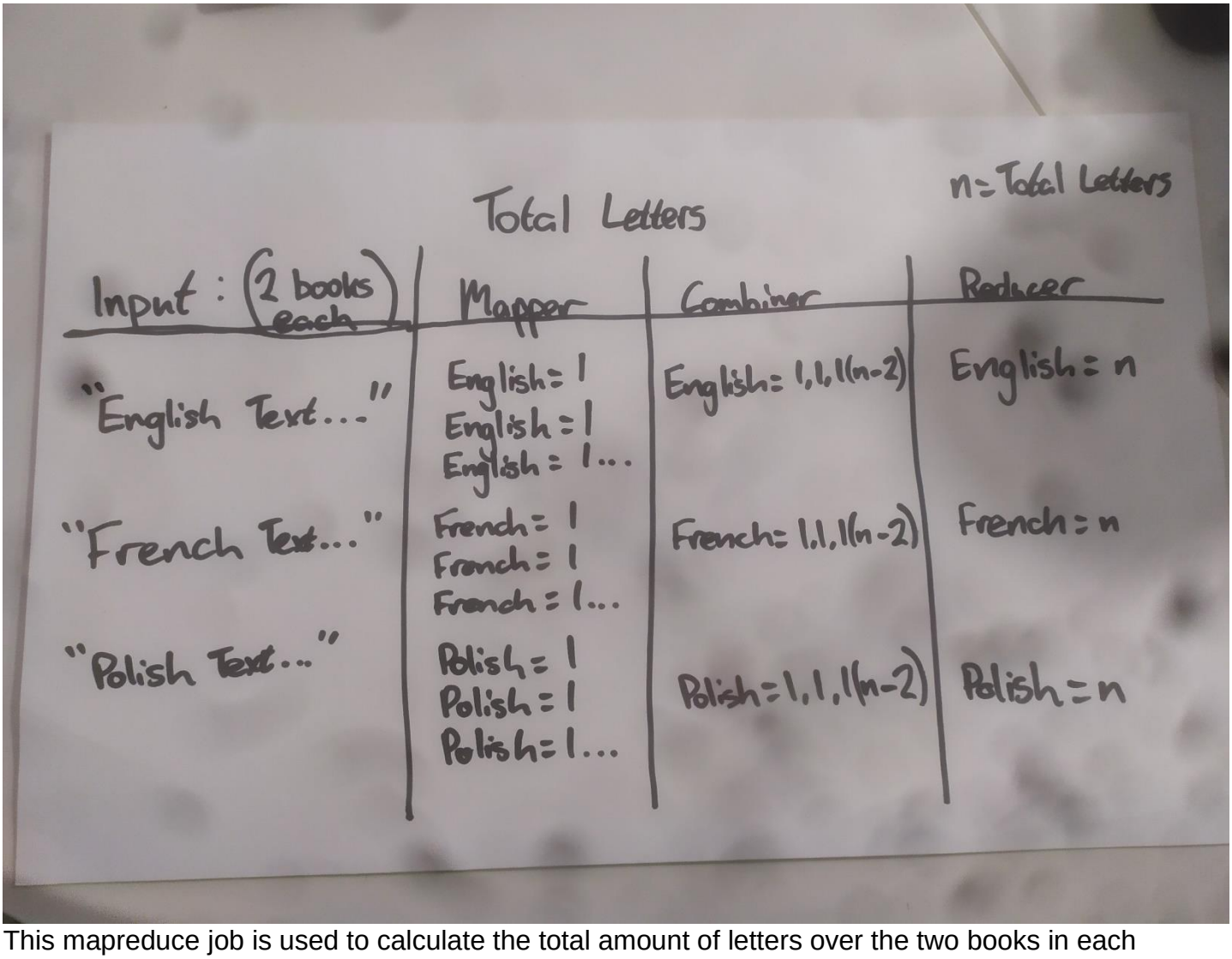
The first step of the design for this process will first involve two mapreduce jobs running in parallel. The first parallel job calculates the total number of letters in each of the three chosen languages (English, French and Polish), the second parallel job calculates the total occurrences of each letter in each language. The second step of this mapreduce process takes the output from the two parallel jobs (*LetterFrequency* and *TotalLetters*) and uses them in one more mapreduce process which calculates the *AverageOccurrences* of each letter.

(AverageOccurences = LetterFrequency/TotalLetters)



Detailed Design:

Job 1: Total Letters



This mapreduce job is used to calculate the total amount of letters over the two books in each language. To do this the **TotalMapper** class takes the six files from the given directory in HDFS (which are named according to the language they are in) as the input. The input is then split into string tokens, all letters are reduced to lower case and special characters and numbers are stripped from the data set so only relevant data is being worked on. Furthermore, a regular expression is added to include accented letters from French and Polish in the data.

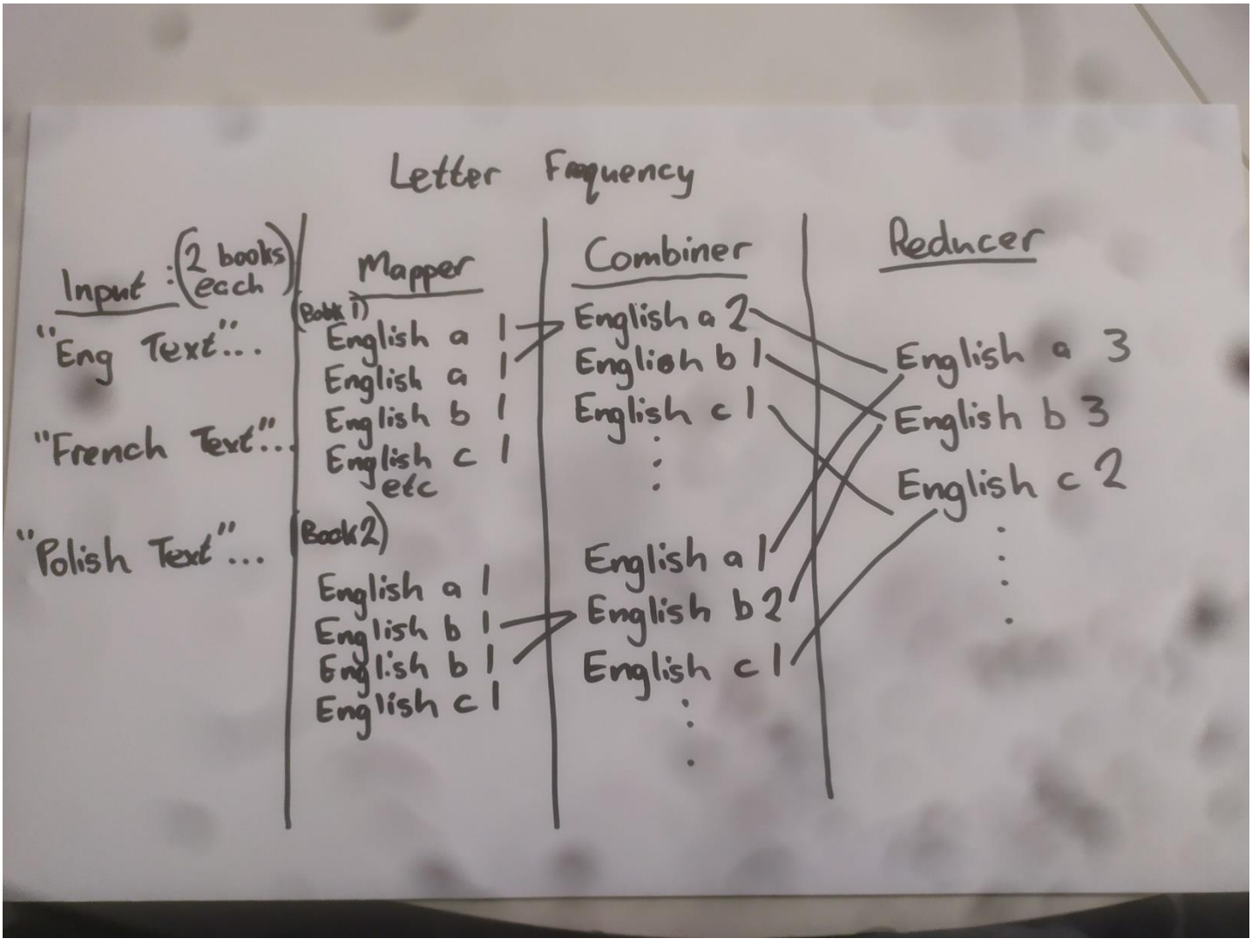
```
StringTokenizer tokenizer = new StringTokenizer(value.toString().toLowerCase().replaceAll("[^\\p{L}]", ""));
```

The names of the input files are also read in order to find the language that each book belongs to which is to be used for the output of the mapper class to later be used by the reducer class.

```
String fileName = ((FileSplit) context.getInputSplit()).getPath().getName();

String language = fileName.replaceAll("[0-9]", "");
```

The string tokens are then looped through and for each token to produce the output of (language = 1), eg ‘English = 1’. The ‘language =’ part of the output is combined using the **TextPair** class which will later be used in the Average Occurrences part of the process. This job makes uses of the combiner feature which alleviates some of the pressure on the reducer and reduces bandwidth when transferring data from the mapper to the reducer. It combines each English = 1 and places it in one line IE <English = 1,1,1…etc> for the **TotalReducer** to calculate the final total.

Job 2: Letter Frequency



This mapreduce job is used to calculate the total amount of each individual letter over the two books in each language. To do this the **FreqMapper** class takes the six files from the given directory in HDFS (which are named according to the language they are in) as the input. The input is then split into string tokens, all letters are reduced to lower case and special characters and numbers are stripped from the data set so only relevant data is being worked on. Furthermore, a regular expression is added to include accented letters from French and Polish in the data. This process is quite like the **TotalMapper** class except it implements the **TextPair** class which was provided as part of the assignment in a different way. This class is used to pair the relevant languages and the associated letter. The string tokens are looped through and for each token to produce the output of (language a 1), (language b 1) etc. The 'language a' part of the output is formed using the **TextPair** class which will later be used in the Average Occurrences part of the process. This job makes uses of the combiner feature which alleviates some of the pressure on the reducer by reducing the

amount of data transfer between the two classes. It combines each English = 1 and places it in one line IE <English = 3> instead of <English =1,1,1> for the **TotalReducer** to calculate the final total.

```
String letter = String.valueOf(string.charAt(x));
TextPair textP = new TextPair(new Text(language), new Text(letter));
 context.write(textP, new IntWritable(1));
```

The **FreqReducer** class then takes the output from the **FreqMapper** class and adds all the 1s together to produce the count for each letter in each language by looping through each of the values and getting a count for each of them.

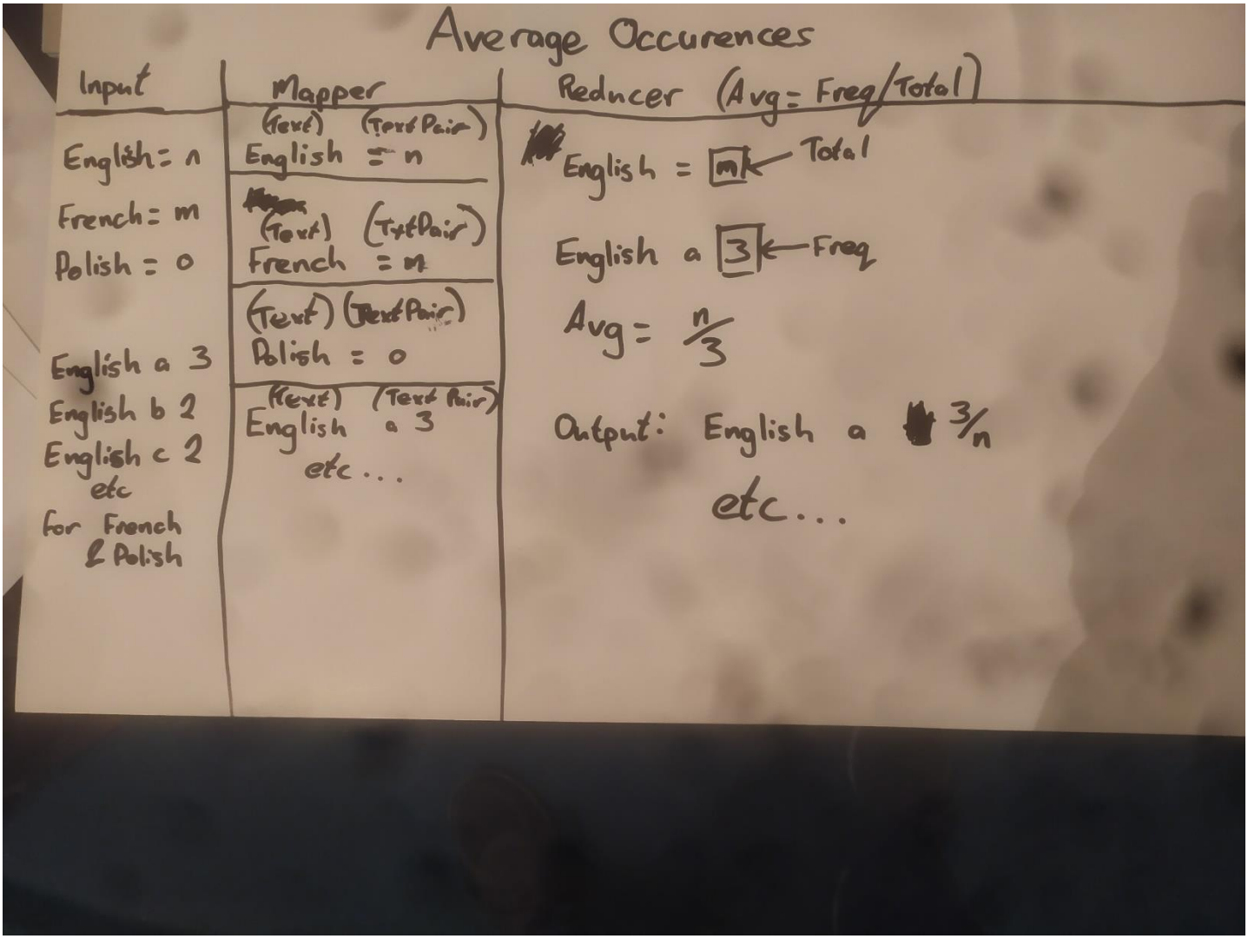
```
int count = 0;
for(IntWritable value: values) {
        count += value.get();
}
```

The output for this MapReduce job is created by pairing the ‘key’, which is the TextPair of (English a), (Polish b), (French é) etc with the count which was found in the previous code example.

```
context.write(key, new IntWritable(count));
```

Job 3: Average Occurrences

This MapReduce job is used to calculate the average occurrence of each individual letter over each language. This is calculated by dividing the frequency of each letter by the total amount of letters in each language which were calculated in the previous two jobs.



To do this the **AverageMapper** class reads the IntWritable output from the TotalLetters and LetterFrequency jobs, splits them into arrays and outputs them in a (Text, TextPair) combination. The **AverageReducer** then loops through all those outputs, if the first element of the textPair is an equal sign ('=') it means that the second element of the TextPair is the TotalLetter count and is set into the variable **count**. If the second element is not an equal sign it means that the second element of the TextPair is the LetterFrequency and is set into the variable **freq.**

```
                int count = 0;

                MarkableIterator<TextPair> markI = new MarkableIterator<TextPair>(textPair.iterator());
                markI.mark();
                while (markI.hasNext()) //inputs are looped through and if first element is equals sign, the second element is the count
                {
                        TextPair textP = markI.next();
                        if(textP.getFirst().toString().equals("=")) { //if there is an equals sign it means the second element of the text pair is the total
                                count = Integer.parseInt(textP.getSecond().toString());
                        }
                }
                markI.reset(); /*reset iterator and loop through them again to find the letter frequency
                inputs are looped through and if first element is not equals sign, the second element is the frequency*/
                while(markI.hasNext())
                {
                        TextPair textP = markI.next();
                        if(!textP.getFirst().toString().equals("=")) { //if there is no equals sign it means the second element of the text pair is the frequency
                                int freq = Integer.parseInt(textP.getSecond().toString()); //frequency of each is set to the number which is located at the second TextPair position
```

The average is then calculated by dividing the frequency by the count. It is formatted to 4 decimal places and an output is written using the key (which is the language) and then a textpair consisting of the letter and rounded average.

```
                                double avg = (double)freq / count ; //get average by dividing the two values
                                DecimalFormat df = new DecimalFormat("#.####");
                                Double roundedAvg = Double.valueOf(df.format(avg)); //format to 4 decimal places

                                context.write(key, new TextPair(textP.getFirst().toString(), String.valueOf(roundedAvg)));
                                //output the key (which is the language), then a TextPair consisting of the letter and average occurrence which is rounded to 4 decimal places
```

Well written and fully commented Java code for the map-reduce process.

TotalMapper.java

```
package Hadoop;


import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class TotalMapper extends Mapper<LongWritable, Text , TextPair, IntWritable> {
        private final static IntWritable one = new IntWritable(1);

        public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
                //regular expression to allow letters from any language and discount
numbers/special chars
                StringTokenizer tokenizer = new
StringTokenizer(value.toString().toLowerCase().replaceAll("[^\\p{L}]", ""));
                // Book files are named according to language and this is how the output
language is extracted
                String fileName = ((FileSplit) context.getInputSplit()).getPath().getName();
                //checks if filename contains the languages and passes it into language variable
for output
                String language = fileName.replaceAll("[0-9]", "");
      while (tokenizer.hasMoreTokens())
      {

        String str = tokenizer.nextToken();
         for(int i = 0; i<str.length(); i++)
```

```
        {
                // equals sign will be used later for extracting its corresponding textpair value
            context.write(new TextPair(language, "="), one);
        }
    }
        }

}
```

TotalReducer.java

```
package Hadoop;

import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.Reducer;

public class TotalReducer extends Reducer<TextPair, IntWritable, TextPair, IntWritable> {

        public void reduce(TextPair key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {
                int languageLetterCount = 0;
                for(IntWritable value: values) {
                        // adds up all the 1s passed by mapper to produce total count of letters
                        languageLetterCount += value.get();
                }
                context.write(key, new IntWritable(languageLetterCount));
        }

}
```

FreqMapper.java

```
package Hadoop;



import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class FreqMapper extends Mapper<LongWritable, Text , TextPair, IntWritable> {

	public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
		//regular expression to allow letters from any language and discount
numbers/special chars
		StringTokenizer tokenizer = new
StringTokenizer(value.toString().toLowerCase().replaceAll("[^\\p{L}]", ""));
		// Book files are named according to language and this is how the output
language is extracted
		String fileName = ((FileSplit) context.getInputSplit()).getPath().getName();

		//checks if filename contains the languages and passes it into language variable
for output
		String language = fileName.replaceAll("[0-9]", "");
	while (tokenizer.hasMoreTokens())
	{

	String string = tokenizer.nextToken();
	 for(int x = 0; x<string.length(); x++)
	 {
		// gets the letter and converts it to string
		String letter = String.valueOf(string.charAt(x));
```

```
                TextPair textP = new TextPair(new Text(language), new Text(letter));
            context.write(textP, new IntWritable(1));
          }
       }
         }
}
```

FreqReducer.java

```
package Hadoop;
import java.io.IOException;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.io.IntWritable;


public class FreqReducer extends Reducer<TextPair, IntWritable, TextPair, IntWritable> {
        //adds all the 1s together that came from the mapper class to produce the frequency of
each letter.
        public void reduce(TextPair key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {

                int count = 0;

                for(IntWritable value: values) {

                        count += value.get();
                }

                context.write(key, new IntWritable(count));
        }
}
```

AverageMapper.java

```
package Hadoop;

import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class AverageMapper extends Mapper<LongWritable, Text , Text, TextPair> {

	public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
		//split each entry from the two sets of inputs into an array,
		String output [] = value.toString().split("\\s");
		//write  array index 0 along with index 1 and 2 in a textpair which will be used for
accessing the total count and the frequency of each letter
		context.write(new Text(output[0]), new TextPair(output[1], output[2]));

	}
}
```

AverageReducer.java

```
package Hadoop;
import java.io.IOException;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.MarkableIterator;
import java.text.DecimalFormat;
import org.apache.hadoop.io.Text;


public class AverageReducer extends Reducer<Text, TextPair, Text, TextPair> {

	public void reduce(Text key, Iterable<TextPair> textPair, Context context) throws
IOException, InterruptedException {
		/*empty value for the total number located at the second item of text pair in
total output
		 * to be iterated over to get total counts for each language
		 * which will then be divided into the frequency of each letter*/
		int count = 0;

		MarkableIterator<TextPair> markI = new
MarkableIterator<TextPair>(textPair.iterator());
		markI.mark();
		while (markI.hasNext()) //inputs are looped through and if first element is equals
sign, the second element is the count
		{
			TextPair textP = markI.next();
			if(textP.getFirst().toString().equals("=")) { //if there is an equals sign it
means the second element of the text pair is the total
				count = Integer.parseInt(textP.getSecond().toString());
			}
		}
		markI.reset(); /*reset iterator and loop through them again to find the letter
frequency
		inputs are looped through and if first element is not equals sign, the second
element is the frequency*/
		while(markI.hasNext())
```

```
                {
                        TextPair textP = markI.next();
                        if(!textP.getFirst().toString().equals("=")) { //if there is no equals sign it
means the second element of the text pair is the frequency
                                int freq = Integer.parseInt(textP.getSecond().toString()); //frequency
of each is set to the number which is located at the second TextPair position
                                double avg = (double)freq / count ; //get average by dividing the two
values
                                DecimalFormat df = new DecimalFormat("#.####");
                                Double roundedAvg = Double.valueOf(df.format(avg)); //format to 4
decimal places

                                context.write(key, new TextPair(textP.getFirst().toString(),
String.valueOf(roundedAvg)));
                                //output the key (which is the language), then a TextPair consisting
of the letter and average occurrence which is rounded to 4 decimal places
                        }
                }
}
```

TextPair.java

```
package Hadoop;

// Sourced from Brendan Tierney's webpage - https://b-tierney.com/msc-prog-big-data/textpairs-code/

import java.io.DataInput;

import java.io.DataOutput;

import java.io.IOException;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.io.WritableComparable;



public class TextPair implements WritableComparable<TextPair> {

private Text first;

private Text second;

public TextPair() {

set(new Text(), new Text());

}

public TextPair(String first, String second) {

set(new Text(first), new Text(second));

}

public TextPair(Text first, Text second) {
```

```
set(first, second);

}

public void set(Text first, Text second) {

this.first = first;

this.second = second;

}


public void setFirst(String first){

this.first = new Text(first);

}


public void setSecond(String second){

this.second = new Text(second);

}


public Text getFirst() {

return first;

}
```

```
public Text getSecond() {

return second;

}

@Override

public void write(DataOutput out) throws IOException {

first.write(out);

second.write(out);

}

@Override

public void readFields(DataInput in) throws IOException {

first.readFields(in);

second.readFields(in);

}

@Override

public int hashCode() {

return first.hashCode() * 163 + second.hashCode();

}

@Override

public boolean equals(Object o) {
```

```
if (o instanceof TextPair) {
TextPair tp = (TextPair) o;
return first.equals(tp.first) && second.equals(tp.second);
}
return false;
}
@Override
public String toString() {
return first + "\t" + second;
}
@Override
public int compareTo(TextPair tp) {
int cmp = first.compareTo(tp.first);
if (cmp != 0) {
return cmp;
}
return second.compareTo(tp.second);
}
}
```

LetterAveragesDriver.java

```
package Hadoop;



import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.jobcontrol.JobControl;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.jobcontrol.ControlledJob;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;


public class LetterAveragesDriver extends Configured implements Tool{
	public static void main(String [] args) throws Exception {
		if(args.length != 3) {
			System.err.println("Usage: LetterAverages <input path> <output path>");
			System.exit(-1);
		}

		int exitCode = ToolRunner.run(new LetterAveragesDriver(), args);
		System.exit(exitCode);

	}

	public int run(String[] args) throws Exception {


```

```
        Configuration conf1 = new Configuration();
        Job TotalLetters = Job.getInstance(conf1);
        TotalLetters.setJarByClass(LetterAveragesDriver.class);
        TotalLetters.setJobName("Total Letters");


        FileInputFormat.setInputDirRecursive(TotalLetters, true);
        FileInputFormat.addInputPath(TotalLetters, new Path(args[1]));
        FileOutputFormat.setOutputPath(TotalLetters, new
Path(args[2]+"/TotalLetters"));

        TotalLetters.setMapperClass(TotalMapper.class);
        TotalLetters.setCombinerClass(TotalReducer.class);
        TotalLetters.setReducerClass(TotalReducer.class);

        TotalLetters.setMapOutputKeyClass(TextPair.class);
        TotalLetters.setMapOutputValueClass(IntWritable.class);

        ControlledJob controlledTotalLetters = new ControlledJob(conf1);
    controlledTotalLetters.setJob(TotalLetters);



    Configuration conf2 = new Configuration();
        Job LetterFrequency = Job.getInstance(conf2);
        LetterFrequency.setJarByClass(LetterAveragesDriver.class);
        LetterFrequency.setJobName("Letter Frequencies");


        FileInputFormat.setInputDirRecursive(LetterFrequency, true);
        FileInputFormat.addInputPath(LetterFrequency, new Path(args[1]));
        FileOutputFormat.setOutputPath(LetterFrequency, new
Path(args[2]+"/LetterFrequency"));

        LetterFrequency.setMapperClass(FreqMapper.class);
        LetterFrequency.setCombinerClass(FreqReducer.class);
        LetterFrequency.setReducerClass(FreqReducer.class);
```

```
        LetterFrequency.setMapOutputKeyClass(TextPair.class);
        LetterFrequency.setMapOutputValueClass(IntWritable.class);

        ControlledJob controlledLetterFrequency = new ControlledJob(conf2);
    controlledLetterFrequency.setJob(LetterFrequency);



    Configuration conf3 = new Configuration();
        Job AverageOccurences = Job.getInstance(conf3);
        AverageOccurences.setJarByClass(LetterAveragesDriver.class);
        AverageOccurences.setJobName("Average Occurrences");

        FileInputFormat.setInputDirRecursive(AverageOccurences, true);
        FileInputFormat.addInputPath(AverageOccurences, new
Path(args[2]+"/TotalLetters"));
        FileInputFormat.addInputPath(AverageOccurences, new
Path(args[2]+"/LetterFrequency"));
        FileOutputFormat.setOutputPath(AverageOccurences, new
Path(args[2]+"/AverageOccurences"));

        AverageOccurences.setMapperClass(AverageMapper.class);
        AverageOccurences.setReducerClass(AverageReducer.class);

        AverageOccurences.setMapOutputKeyClass(Text.class);
        AverageOccurences.setMapOutputValueClass(TextPair.class);

        ControlledJob controlledAverageOccurences = new ControlledJob(conf3);
    controlledAverageOccurences.setJob(AverageOccurences);


    controlledAverageOccurences.addDependingJob(controlledTotalLetters); //Average
Occurrences job cannot start until Total Letters and Letter Frequency have finished
    controlledAverageOccurences.addDependingJob(controlledLetterFrequency);
```

```
        JobControl jobCtrl = new JobControl("jobcontrol"); //create job control to direct
the workflow of each job
        jobCtrl.addJob(controlledAverageOccurences);
        jobCtrl.addJob(controlledLetterFrequency);
        jobCtrl.addJob(controlledTotalLetters);


        Thread jobControlThread = new Thread(jobCtrl);
        jobControlThread.start();

        while (!jobCtrl.allFinished()) {
            System.out.println("Executing jobs, still running...");
            Thread.sleep(5000);
        }
            System.out.println("Finished Executing, check the Hadoop filesystem!");
            jobCtrl.stop();
          return (AverageOccurences.waitForCompletion(true) ? 0 : 1);
      }

}
```

Advanced Map-Reduce functions

Combiners

Combiners were used in two of the MapReduce jobs in this process, namely the Total Letters and Letter Frequency jobs. They were implemented in the driver class with the following lines of code:

```
LetterFrequency.setCombinerClass(FreqReducer.class);

TotalLetters.setCombinerClass(TotalReducer.class);
```

I did not feel it was necessary to write specific Combiner classes for these jobs so set the Combiner class to use the same code as the Reducer class for each job. The purpose of using the combiners in these jobs is to increase efficiency. Part of the reducing phase is already done before the data is sent from the mapper to the reducer. This reduces the amount of data sent which uses less bandwidth and in turn improves the speed at which the Mapper completes its job due to it having less work to do.

Chaining Multiple Jobs

Three MapReduce jobs were written for this process. The first two jobs – Total Letters and Letter Frequency – were run in parallel with each other to reduce processing time.

To direct the workflow of the three jobs, they each needed to be defined as controlled jobs within the driver.

```
ControlledJob controlledTotalLetters = new ControlledJob(conf1);
controlledTotalLetters.setJob(TotalLetters);

ControlledJob controlledLetterFrequency = new ControlledJob(conf2);
controlledLetterFrequency.setJob(LetterFrequency);

ControlledJob controlledAverageOccurences = new ControlledJob(conf3);
controlledAverageOccurences.setJob(AverageOccurences);
```

The third job requires the outputs from the first two jobs in order for it to start so it was set up to start only once they were finished. This was done by defining dependencies in the driver class:

```
controlledAverageOccurences.addDependingJob(controlledTotalLetters);
controlledAverageOccurences.addDependingJob(controlledLetterFrequency);
```

A job control was created to direct the workflow and the three controlled jobs were added to it:

```
JobControl jobCtrl = new JobControl("jobcontrol");
jobCtrl.addJob(controlledAverageOccurences);
jobCtrl.addJob(controlledLetterFrequency);
jobCtrl.addJob(controlledTotalLetters);
```

Finally, to run the series of jobs in the correct order a Thread is defined and is set to the variable name of the JobControl 'jobCtrl'. Once this is done the thread of controlled jobs can be started.

```
Thread jobControlThread = new Thread(jobCtrl);
jobControlThread.start();
```

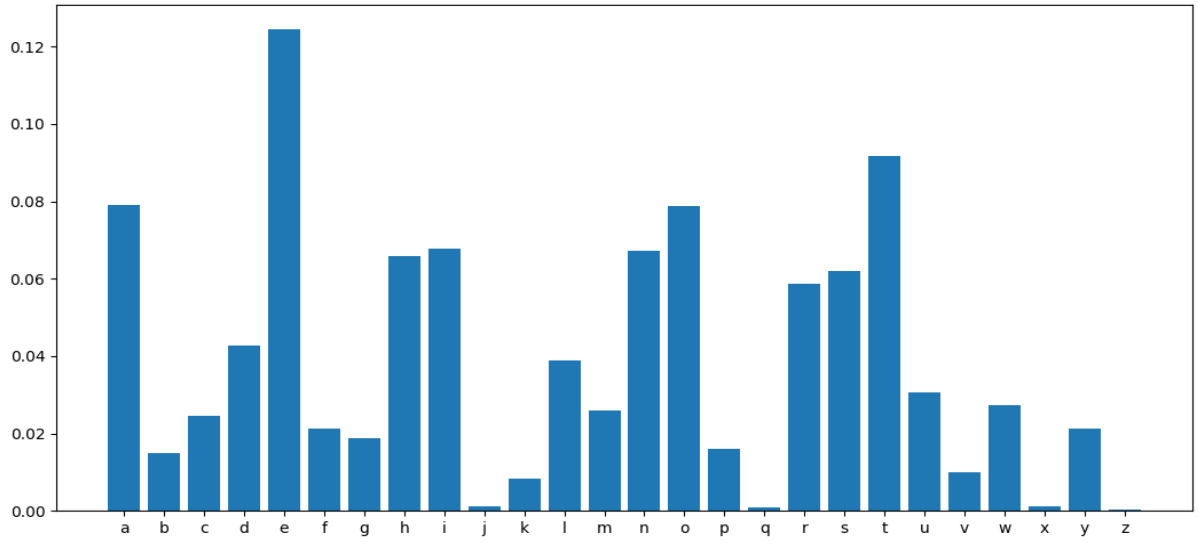
R/Python code used to load the data sets and create the comparison charts. You should provide some commentary for each of the charts included in your report.

Code:

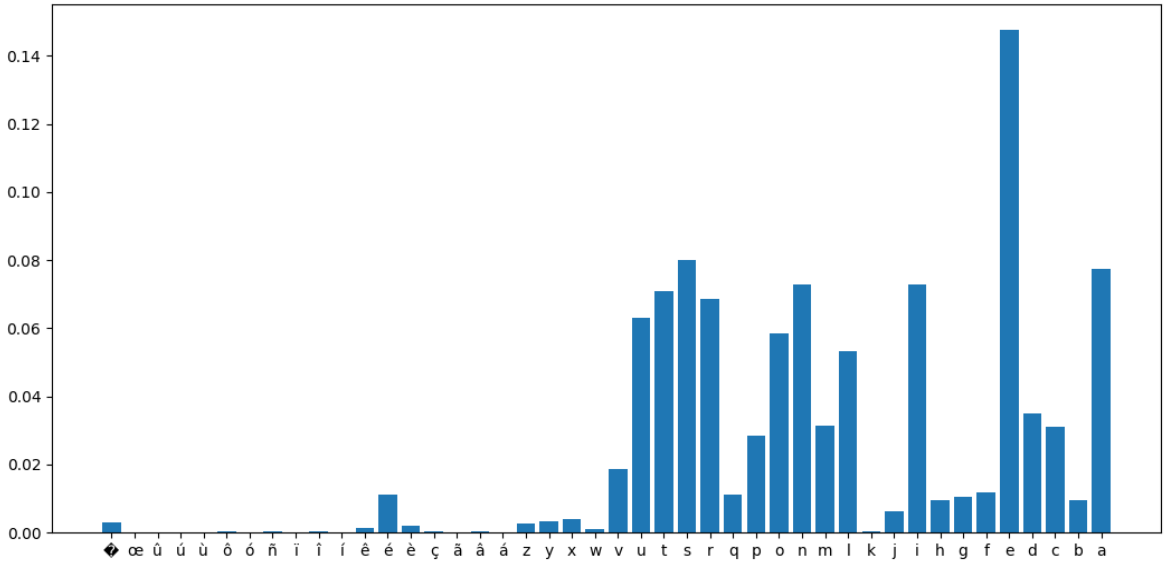
```
import matplotlib as mpl
import matplotlib.pyplot as plt
import pandas as pd
import csv

#convert txt file to csv
with open('AverageOccurrences.txt') as infile, open('AverageOccurrences.csv', 'w') as outfil
e:
    for line in infile:
        outfile.write(" ".join(line.split()).replace(' ', ','))
        outfile.write(",") #seems to add an extra emplty column
        outfile.write("\n")

df = pd.read_csv("AverageOccurrences.csv", header =None) # read the text file which has no h
eader so first line is not read as column name

df.columns =['Language', 'Letter', 'Frequency', ' '] #set column names


#split into separate csv files according to language column values
csv_english = df[df['Language'] == 'English']
csv_french = df[df['Language'] == 'French']
csv_polish = df[df['Language'] == 'Polish']

eng= csv_english.to_csv('english.csv', index=False, header= None)
fr=csv_french.to_csv('french.csv', index=False, header= None)
pl=csv_polish.to_csv('polish.csv', index=False, header= None) # split into separate csvs


eng = pd.read_csv("english.csv", header =None)
fr = pd.read_csv("french.csv", header =None)
pl = pd.read_csv("polish.csv", header =None)

eng.columns =['Language', 'Letter', 'Frequency', ' '] #set column names for the new csv file
s
```

```
fr.columns =['Language', 'Letter', 'Frequency', ' ']
pl.columns =['Language', 'Letter', 'Frequency', ' ']



################################### MATPLOTLIB PART ###################################

engx = eng['Letter'].tolist()
engy = eng['Frequency'].tolist()
plt.bar(engx,engy)
plt.show()

frx = fr['Letter'].tolist()
fry = fr['Frequency'].tolist()
plt.bar(frx,fry)
plt.show()

plx = pl['Letter'].tolist()
ply = pl['Frequency'].tolist()
plt.bar(plx,ply)
plt.show()
```

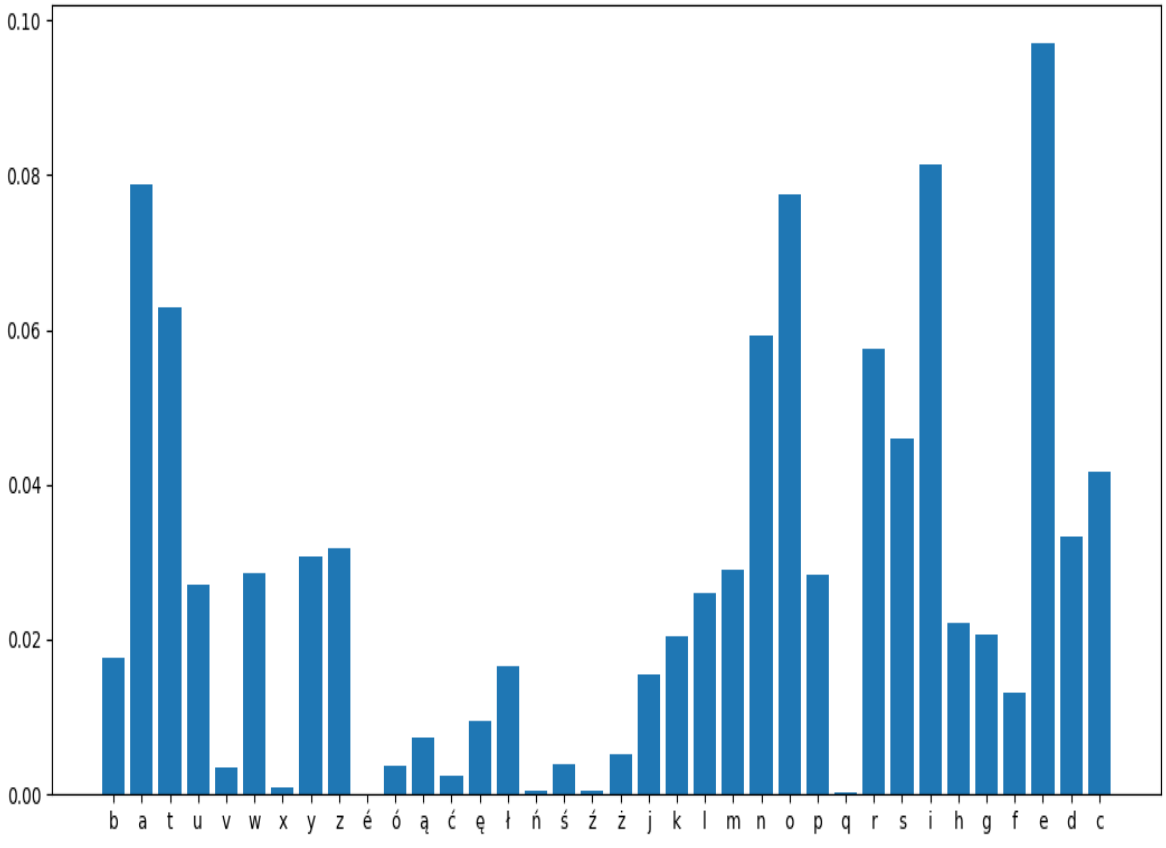
English Chart:



French Chart:



Polish Chart:



From studying these charts, it is clear to see that some letters stand out as the most common across each of the languages. Four out of the five most common letters in English are vowels, three out of the five in French and four out of five in Polish. This is most likely due to the very limited number of words in the English language that do not contain vowels and because there are only five vowels in any language making their use very common. The letters j, q, x and z are clearly the least common letters in all languages and accented letters which only apply to French and Polish seem to appear very seldom. The letter ‘z’ which appears very rarely in both English and French is surprisingly common in Polish and occurs in nearly 3% of the letter occurrences which is quite high considering the most common letter in any language surveyed in this study appears 14% of the time (‘e’ in French) and the most common letter in Polish appears in less than 10% of the letter occurrences (‘e’ in Polish).