<u>The assignment:</u>

Project context: (imaginary)

- You work within a team of 4 developers (yourself, another ML engineer and two web devs).
- You are asked to build the basics of a long term project, your code needs to be easily understood and reusable
- Your code does not need to be optimal in terms of performance but needs to be fast enough to provide a good user experience (the input data might scale to millions of records at some point)
- Your code needs to be trusted it does what it is meant to do
- At the end, your code will be peer reviewed and released on Github

Gutenberg Words

The Spark Batch Challenge consists on analyzing the text of the Gutenberg Project using Apache Spark as a batch processing engine.

Download the data

This challenge uses the data of the Gutenberg Project in txt-format.

To download the complete content of the Gutenberg online library run (bash):

```
curl -sSL https://raw.githubusercontent.com/RHDZMOTA/spark-wordcount/develop/gutenberg.sh | sh
```

Depending on your network speed this can take up to 3 hours.

You can see the "footprint" of the content with:

- Number of books: `ls -l gutenberg | wc -l`
- Data size: `du -sh gutenberg`

We will work with a random sample of 5000 books. Run the following command to create the sample:

```
mkdir gutenberg-sample && ls gutenberg/ | shuf -n 5000 | xargs -I _ cp
gutenberg/_ gutenberg-sample/_
```

Problem definition

In this section we define the scope and tasks for each problem.

Each problem is evaluated sequentially due to possible dependency with advanced tasks of further problems. The reviewer will evaluate the output for each problem, clean code and best practices. The following convention will be used:

The solution of problem `i` should be executed when running:

```
sbt "runMain com.intersys.mdc.challenge.spark.batch.problem<1>.Solution"
```

Where:

```
   i : is the number of the problem (i.e., 1, 2, 3)
```

Example The solution of the first problem should execute when running:

- `sbt "runMain com.intersys.mdc.challenge.batch.problem1.Solution"`

Part I : Zipf's Law

The Zipf's law is an empirical law that states that for a large sample of words, the frequency of a given word is inversely proportional to the rank on the word-count table. Therefore, a given word with rank `i` has a frequency proportional to `1/i`. This second most frequent word will occur about half the times (1/2) of the first most frequent word.

In this problem you'll have e to test the Zipf's Law using the words contained on the books of the Gutenberg Project.

Keypoints

- Use the Apache Spark Scala API.
  - Demonstrate understanding of the Dataset[T] datatype.
  - Use Scala's `case class` to represent domain models.
- Output file should be a single csv-file contained in `resources/output/zipf`.
  - Must have the following columns: `word`, `count`, `frequency`, `zipf`

    - Must be sorted (descending) by the `count` column.
- Running the application
    - `sbt "runMain com.intersys.mdc.challenge.spark.batch.problem1.Solution"`

Columns:

```
   word      : a column containing the unique words in the gutenberg-sample folder.
    count     : the number of appearance of a word in the project.
    frequency : count / total_unique_words
    rank      : position of the word when the dataset is ordered.
    zipf      : estimation fo the zipfian distribution (i.e. 1 / rank)
```

Part 2 : Word Prediction Database

In this problem you'll have to populate a mongodb database from Apache Spark. The goal is to create a collection that contains each unique word and the probable following words.

Example:

```
Consider the following text as the complete knowledge-base: "a b c a b c d e a a c"
```

We know that the `a` word is followed by `b`, `a`, and `c`. We can represent this with the following json:

```
{
  "word": "a",
  "next": [
    {"nw": "b", "p": 0.5},
    {"nw": "a", "p": 0.25},
    {"nw": "c", "p": 0.25}
  ]
}
```

Where:

```
   word : Represent a word in the Gutenberg Project sample.
    next : Is a list of possible following words.
    nw   : Is a possible "following word" for the original "word".
    p    : Is the probability of finding nw as decimal (note that the
sum of all "p" must be 1.0).
```

Keypoints

- Using the `gutenberg-sample` data create a `Dataset[WordRegister]` where `WordRegister` is the following case class:

  ```scala
  final case class WordRegister(word: String, next: List[Word.NextWord])

  object WordRegister {
    final case class NextWord(nw: String, p: Double)
  }
  ```

- Use the library of your choice (e.g. circe) to be able to represent the a `WordRegister` instance as a json-string:

  ```json
  {
    "word": "some-word",
    "next": [
      {"nw": "next-word-1", "p": 0.7},
      {"nw": "next-word-2", "p": 0.2},
      {"nw": "next-word-3", "p": 0.1} ]
  }
  ```

- Populate a MongoDB collection with the data contained on the `Dataset[WordRegister]`.
  - See MongoDB Spark Connector
    - Database name `gutenberg`
    - Collection `words`
  - Use Docker to create a container with a Mongodb image.
    - `docker run -d -p 27017:27017 mongo`
- Running the application should perform the Spark transformation and populate the mongodb database.
  - `sbt "runMain com.intersys.mdc.challenge.spark.batch.problem2.Solution"`
  - You can query your results form the command line:
    - `sudo apt install mongodb-clients`
    - `mongo <ip-address>/gutenberg`
    - `db.words.find()`

Part 3: Word Prediction Engine

Using the resulting mongodb collection from the previous problem you'll have to create a REST API to predict the probable following words given an initial word.

Keypoints

- Create a REST API using Scala (recommended: Akka HTTP).
    - Connect to mongodb using the official Scala Driver
- The API should expose the following endpoints:
    - Next possible words
        - Get request: `http://localhost:8080/gutenberg/predict/next?word="a"`
        - Response: `{"words": ["b", "a", "c"], "probability": [0.5. 0.25, 0.25]}`
    - Random guess
        - Get request: `http://localhost:8080/gutenberg/predict/random?word="a"`
        - Response: `{"guess": "a"}`
- Running the application the API should be up and running after the following command.
    - `sbt "runMain "com.intersys.mdc.challenge.spark.batch.problem3.Solution`