Assignment 1: Sarcasm Analysis

Due Date: 11/2

Submit your questions to Gal

Abstract

In this assignment you will code a real-world application to distributively process a list of amazon reviews, perform *sentiment analysis* and *named-entity recognition*, and display the result on a web page. The goal of the assignment is to experiment with AWS and detect sarcasm!

More Details

The application is composed of a *local application*, and *non-local instances* running on Amazon cloud.

The application gets input text files containing lists of reviews (JSON format).

Then, instances are launched in AWS (workers & a manager) to apply sentiment analysis on the reviews and detect whether it is sarcasm or not. The results are displayed on a webpage.

The main use-case:

1. The user starts the application and supplies as input files with reviews, an integer *n* stating how many reviews/messages per worker (your choice), and an optional argument *terminate*, indicating that the local application will send a terminate message to the Manager at the end of its task.
2. The user gets an html file containing the reviews with sentiment analysis and sarcasm detection, containing the url of the original review (at amazon)

Input Files

For this assignment we provide five input files, in JSON format (use any kit to parse the input).

Output File

The output is an HTML file containing a line for each input review:

- A link to the original review, coloured according to its sentiment:
  - dark red: very negative
  - red: negative
  - black: neutral
  - light green: positive
  - dark green: very positive
- A list of the named entities found in the review (comma separated items with under [])
- Sarcasm Detection
  We apply a simple detection algorithm, based on the number of stars given by the user are suitable for the review sentiment analysis. If it is so, then there is no sarcasm, otherwise it appears to be sarcasm.

System Architecture

The system is composed of 3 elements:

- Local application
- Manager
- Workers

The elements communicate with each other using queues (SQS) and storage (S3). It is up to you to decide how many queues to use and how to split the jobs among the workers, but, and you will be graded accordingly, your system should strive to work in **parallel**. It should be as **efficient** as possible in terms of time and money, and **scalable**.

Local Application

The application resides on a local (non-cloud) machine. Once started, it reads the input file from the user, and:

- Checks if a Manager node is active on the EC2 cloud. If it is not, the application will start the manager node.
- Uploads the file to S3.
- Sends a message to an SQS queue, stating the location of the file on S3
- Checks an SQS queue for a message indicating the process is done and the response (the summary file) is available on S3.
- Create an html file representing the results.
- In case of *terminate* mode (as defined by the command-line argument), sends a termination message to the Manager.

IMPORTANT: There might be more than one than one local application running at the same time.

The Manager

The manager process resides on an EC2 node. It checks a special SQS queue for messages from local applications. Once it receives a message it:

- For the case of new task message:
    - Distribute the operations to be performed on the reviews to the workers using SQS queue/s.
    - Check the SQS message count and starts Worker processes (nodes) accordingly.
        - The manager should create a worker for every *n* messages (as defined by the command-line argument), if there are no running workers.
        - If there are *k* active workers, and the new job requires *m* workers, then the manager should create *m*-*k* new workers, if possible.
          For any case don't run more than 19 instances – AWS uses to block students who try this.
        - Note that while the manager creates a node for every *n* messages, it does not delegate messages to specific nodes. All of the worker nodes take their messages from the same SQS queue; so it might be the case that with *2n* messages, hence two worker nodes, one node processed *n+(n/2)* messages, while the other processed only *n/2*.

    - In case the manger receives response messages from the workers (regarding input file), it:
        - Creates a summary output file accordingly
        - Uploads the output file to S3
        - Sends a message to the application with the location of the file
- In case of a termination message, the manager:
    - Should not accept any more input files from local applications. However, it does serve the local application that sent the termination message.
    - Waits for all the workers to finish their job, and then terminates them.
    - Creates response messages for the jobs, if needed.
    - Terminates.

IMPORTANT: the manager must process requests from local applications simultaneously; meaning, it must not handle each request at a time, but rather work on all requests in parallel.

The Workers

A worker process resides on an EC2 node. His life cycle:

Repeatedly:

- Get a message from an SQS queue.
- Perform the requested job, and return the result.
- Remove the processed message from the SQS queue.

IMPORTANT: If a worker stops working unexpectedly before finishing its work on a message, then some other worker should be able to handle that message.

Queues and Messages

As described above, queues are used for:

- Communication between the local application and the manager.
- Communication between the manager and the workers.

It is up to you to decide what the jobs and the messages are, and how many queues to use. Your system should run as efficiently as possible in terms of time.

Running the Application

The application should be run as follows:

```
>java  -jar yourjar.jar inputFileName1... inputFileNameN outputFileName1... outputFileNameN n
[terminate]
```

The worker should run as follows:

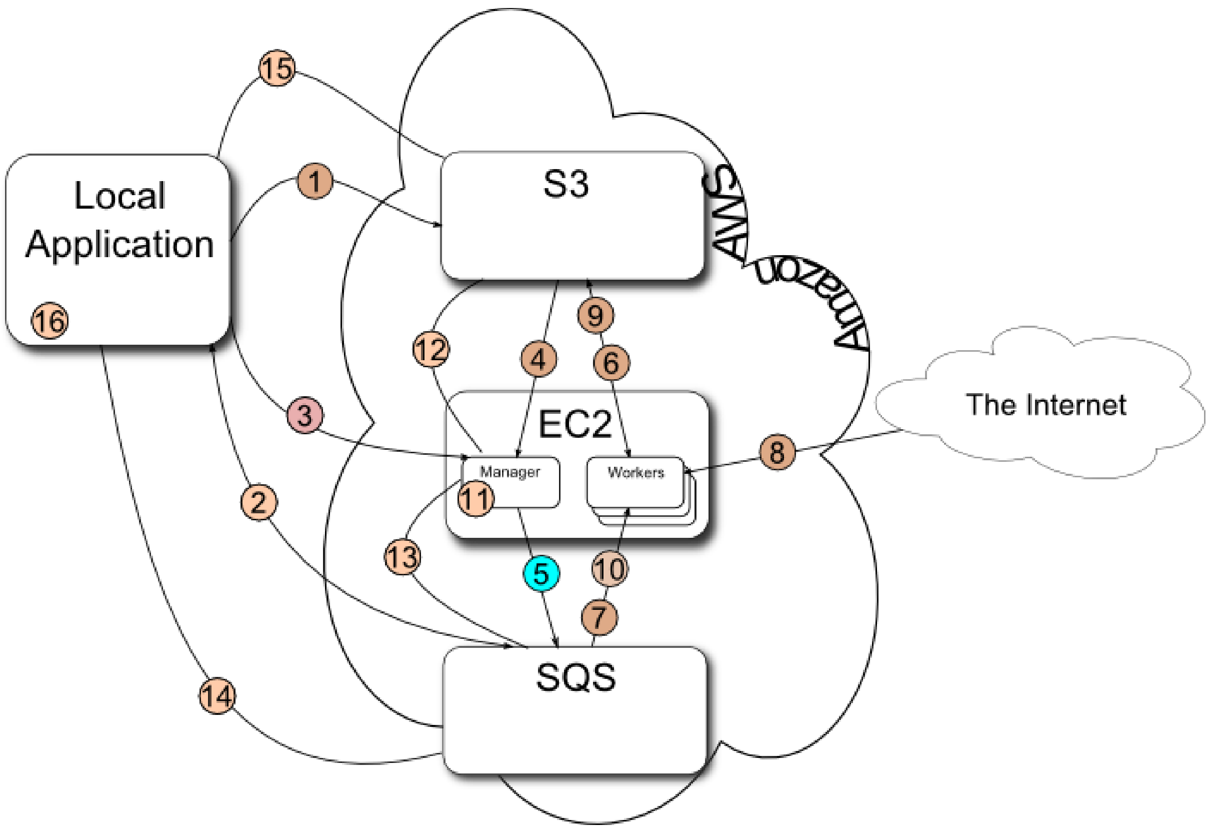
```
>java -jar yourjar.jar MainWorkerClass
```

where:

- *yourjar.jar* is the name of the jar file containing your code (do not include the libraries in it when you create it).
- *MainWorkerClass* is the name of the class with the main method of the Worker code.\\
- *inputFileName$_I$* is the name of the input file *I*.

- *outputFileName* is the name of the output file.
- *n* is the workers' files ratio (reviews per worker).
- *terminate* indicates that the application should terminate the manager at the end.

System Summary



1. Local Application uploads the file with the list of reviews urls to S3.
2. Local Application sends a message (queue) stating the location of the input file on S3.
3. Local Application does one of the following:
   - Starts the manager.
   - Checks if a manager is active and if not, starts it.
4. Manager downloads a list of reviews.
5. Manager distributes sentiment analysis and entity extraction jobs on the workers.
6. Manager bootstraps nodes to process messages.
7. Worker gets a message from an SQS queue.
8. Worker performs the requested job/s on the review.
9. Worker puts a message in an SQS queue indicating the original reviewtogether with the output of the operation performed (sentiment/extracted entities).
10. Manager reads all Workers' messages from SQS and creates one summary file.
11. Manager uploads the summary file to S3.
12. Manager posts an SQS message about the summary file.
13. Local Application reads SQS message.
14. Local Application downloads the summary file from S3.
15. Local Application creates html output file.

16. Local application send a terminate message to the manager if it received <i>terminate</i> as one of its arguments.

Getting Started

- Read the assignment description.
- Read the reading the material, and make sure you understand it.
- Create a maven project.
- Write the code that analyzes a given text (NER, Sentiment), run it on your computer and make sure it works.
- Write the local application code and make sure it works.
- Write the manager code, run it on your computer and make sure it works.
- Run the manager, the local application, together with a worker on your computer and make sure they work.
- Run the local application on your computer, and let it start and run the manager on EC2, and let the manager start and run the workers on EC2.

Technical Stuff

- Getting started with AWS:
    - Read the guide in the AWS academy lab - all you need to know is presented there, and is updated to the latest features and libraries of AWS.
    - Make sure that you place the .pem (credentials) file in the right location, and where your AWS CLI expects it to be -- this is up to you to make sure you do it correctly. Otherwise, you won't be able to communicate with the AWS services.
    - Be aware of AWS regions - make sure that the time of the region you're using (the region of the instances) is consistent with your system (the region of your AWS console, the region of the AWS dashboard). Since the session key of AWS academy is only valid for several hours, large time differences can mean that while your access keys are indeed 'fresh', it is not recognized by AWS.
      This link may be useful.

- AMI Image

A virtual machine (VM) is a software implementation of a machine (i.e. a computer) that executes programs like a physical machine. In our case: EC2 instance. An image of a virtual machine is (in simple words) a copy of the VM, which may contain an OS, data files, and applications (just like your personal computer) - in our case: Amazon Machine image (AMI).

You need an AMI composed of any data needed for running the Manager and the Workers (basically, Linux, Java JDK 1.8, Amazon AWS cli, the jar file of the Manager and the Worker classes, and the OCR jar).

You can create a new AMI from the Amazon EC2 Instances view, or from the console of an EC2 instance:

Downloading from EC2 console: In Linux, the command *wget* is usually installed. You can use it to download web files from the shell.
Example:

wget https://www.guteiageimagnberg.org/files/1659/1659-0.txt -O a.txt
will download the content at https://www.gutenberg.org/files/1659/1659-0.txt and save it to a file named a.txt. wget man

Installing from EC2 console: In Ubuntu (or Debian) Linux, you can use the apt-get command (assuming you have root access to the machine). Example: apt-get install *wget* will install the *wget* command if it is not installed. You can use it to install Java, or other packages. apt-get man.

In case AWS disable the option for creating new AMI, use the user-data script for downloading the required jars each time the node is initialized.

- The AWS SDK

The assignment will be written in Java, you'll need the SDK for Java for it.

AWS Maven dependencies:

```
<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>ec2</artifactId>
  <version>2.22.12</version>
</dependency>

<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>s3</artifactId>
  <version>2.22.12</version>
</dependency>

<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>sqs</artifactId>
  <version>2.22.12</version>
</dependency>
```

- SQS Visibility Time-out

Read the following regarding SQS timeout, understand it, and use it in your implementation: Visibility Timeout

- Bootstrapping

When you create and boot up an EC2 node, it turns on with a specified image, and that's it. You need to load it with software and run the software in order to do something useful. We refer to this as "bootstrapping" the machine.
As taught in class, the bootstrapping process of a new node is based on user data scripts, and another guide. User-data allows you to pass a shell-script to the machine, to be run when it starts up. Notice that the script you're passing should be encoded to base64.
Your user-data scripts can be written in any language you want (e.g. Python, Perl, tsch, bash). bash is

a very common choice. Your scripts are going to be very simple. Nonetheless, you might find these bash tutorials useful.

- Checking if a Manager Node is Active

You can check if the manager is running by listing all your running instances, and checking if one of them is a manager. Use the "tags" feature of Amazon EC2 API to mark a specific instance as one running a Manager: using tags, CreateTagsRequest API

- Third-party libraries

To analyze the reviews, we are going to use Stanford CoreNLP. We are going to perform 2 types of analysis:

Sentiment Analysis: Given a text find out its sentiment; whether what the text is saying is positive/negative/neutral. The tool gives a score between 0 \= very negative up to 4 \= very positive.

Named Entity Extraction: Given a text extracts the entities of the text together with their entity type (e.g. Obama:Person)

Dependencies

Add the following dependencies to the Worker pom.xml file:

```
<dependency>
  <groupId>edu.stanford.nlp</groupId>
  <artifactId>stanford-corenlp</artifactId>
  <version>4.4.0</version>
</dependency>
<dependency>
  <groupId>edu.stanford.nlp</groupId>
  <artifactId>stanford-corenlp</artifactId>
  <version>4.4.0</version>
  <classifier>models</classifier>
</dependency>
<dependency>
  <groupId>de.jollyday</groupId>
  <artifactId>jollyday</artifactId>
  <version>0.5.10</version>
</dependency>
<dependency>
  <groupId>com.googlecode.efficient-java-matrix-library</groupId>
  <artifactId>ejml</artifactId>
  <version>0.23</version>
</dependency>
```

Note: The StanfordCoreNLP constructor needs a lot of resources and takes ~1 Min to build StanfordCoreNLP instance. There for you should have 2 classes (sentimentAnalysisHandler and namedEntityRecognitionHandler) that wrap the code below.

Then you will have them static at the worker Main class:

```
public class Worker {
   static sentimentAnalysisHandler sentimentAnalysisHandler = new      sentimentAnalysisHandler();
   static namedEntityRecognitionHandler namedEntityRecognitionHandler = new
namedEntityRecognitionHandler();
```

Sentiment Analysis

Note: The methods do not have to be static.

Import List:

```
import java.util.List;
import java.util.Properties;
import edu.stanford.nlp.ling.CoreAnnotations;
import edu.stanford.nlp.ling.CoreAnnotations.NamedEntityTagAnnotation;
import edu.stanford.nlp.ling.CoreAnnotations.SentencesAnnotation;
import edu.stanford.nlp.ling.CoreAnnotations.TextAnnotation;
import edu.stanford.nlp.ling.CoreAnnotations.TokensAnnotation;
import edu.stanford.nlp.ling.CoreLabel;
import edu.stanford.nlp.pipeline.Annotation;
import edu.stanford.nlp.pipeline.StanfordCoreNLP;
import edu.stanford.nlp.rnn.RNNCoreAnnotations;
import edu.stanford.nlp.sentiment.SentimentCoreAnnotations;
import edu.stanford.nlp.trees.Tree;
import edu.stanford.nlp.util.CoreMap;
```

Initialization:

```
Properties props = new Properties();
props.put("annotators", "tokenize, ssplit, parse, sentiment");
StanfordCoreNLP  sentimentPipeline =  new StanfordCoreNLP(props);
```

The following code performs sentiment analysis on a review:

```
public static int findSentiment(String review) {
                    int mainSentiment = 0;
                    if (review!= null && review.length() > 0) {
                              int longest = 0;
                              Annotation annotation = sentimentPipeline.process(review);
                              for (CoreMap sentence : annotation
                                                  .get(CoreAnnotations.SentencesAnnotation.class)) {
                                        Tree tree = sentence.get(
                                                  SentimentCoreAnnotations.AnnotatedTree.class);
                                        int sentiment = RNNCoreAnnotations.getPredictedClass(tree);
                                        String partText = sentence.toString();
                                        if (partText.length() > longest) {
                                                  mainSentiment = sentiment;
                                                  longest = partText.length();
                                        }

                              }
                    }
                    return mainSentiment;
}
```

Named Entity Recognition

Extract only the following entity types: PERSON, LOCATION, ORGANIZATION.

Initialization:

```
Properties props = new Properties();
props.put("annotators", "tokenize , ssplit, pos, lemma, ner");
StanfordCoreNLP NERPipeline =  new StanfordCoreNLP(props);
```

The following code extracts named entities from a review:

```
public static void printEntities(String review){
          // create an empty Annotation just with the given text
          Annotation document = new Annotation(review);

          // run all Annotators on this text
          NERPipeline.annotate(document);

          // these are all the sentences in this document
          // a CoreMap is essentially a Map that uses class objects as keys and has values with
custom types
          List<CoreMap> sentences = document.get(SentencesAnnotation.class);

          for(CoreMap sentence: sentences) {
               // traversing the words in the current sentence
               // a CoreLabel is a CoreMap with additional token-specific methods
               for (CoreLabel token: sentence.get(TokensAnnotation.class)) {
                    // this is the text of the token
                    String word = token.get(TextAnnotation.class);
                    // this is the NER label of the token
                    String ne = token.get(NamedEntityTagAnnotation.class);
                    System.out.println("\t-" + word + ":" + ne);
               }
          }

     }
```

Grading

- The assignment will be graded in a frontal setting.
- All information mentioned in the assignment description, or learnt in class, is mandatory for the assignment.

- You will be reduced points for not reading the relevant reading material, not implementing the recommendations mentioned there, and not understanding them.
- Students belonging to the same group will not necessarily receive the same grade.
- All the requirements in the assignment will be checked, any missing functionality will cause a point reduction. Any additional functionality will compensate for lost points. You have the "freedom" to choose how to implement things that are not precisely defined in the assignment.

Notes

Cloud services are cheap but not free. Even if they were free, waste is bad. Therefore, please keep in mind that:

- It should be possible for you to easily remove all the things you put on S3. You can do that by putting them in a specified bucket under a folder, which you could delete later.
- While it is the Manager's job to turn off all the Worker nodes, do verify yourself that all the nodes really did shut down, and turn of the manger manually if it is still running.
- You won't be able to download the files unless you make them public.
- You may assume there will not be any race conditions; conditions where 2 local applications are trying to start a manger at the same time etc.

A Very Important Note about Security

As part of your application, you will have a machine on the cloud contacting an amazon service (SQS and S3). For the communication to be successful, the machine will have to supply the service with a security credentials (password) to authenticate. Security credentials is sensitive data – if someone gets it, they can use it to use amazon services for free (from your budget). You need to take good care to store the credentials in a secure manner. One way of doing that is by compressing the jar files with a password.

Submission

Submit a zip file, contains:

- all sources and class files, without the libraries that you're supposed to download;
- the output of running your system on the sample file.
- a text file called README with instructions on how to run your project, and an explanation of how your program works. It will help you remember your implementation. Your README must also contain what type of instance you used (ami + type:micro/small/large…), how much time it took your program to finish working on the input files, and what was the n you used.

Mandatory Requirements

- Be sure to submit a README file. Does it contain all the requested information? If you miss any part, you will lose points. Don't forget your names and ids…
- Did you think for more than 2 minutes about security? Do not send your credentials in plain text!
- Did you think about scalability? Will your program work properly when 1 million clients connected at the same time? How about 2 million? 1 billion? Scalability is very important aspect of the system; be sure it is scalable!
- What about persistence? What if a node dies? What if a node stalls for a while? Have you taken care of all possible outcomes in the system? Think of more possible issues that might

arise from failures. What did you do to solve it? What about broken communications? Be sure to handle all fail-cases!

- Threads in your application, when is it a good idea? When is it bad? Invest time to think about threads in your application!
- Did you run more than one client at the same time? Be sure they work properly, and finish properly, and your results are correct.
- Do you understand how the system works? Do a full run using pen and paper, draw the different parts and the communication that happens between them.
- Did you manage the termination process? Be sure all is closed once requested!
- Did you take in mind the system limitations that we are using? Be sure to use it to its fullest!
- Are all your workers working hard? Or some are slacking? Why?
- Is your manager doing more work than he's supposed to? Have you made sure each part of your system has properly defined tasks? Did you mix their tasks? Don't!
- Lastly, are you sure you understand what distributed means? Is there anything in your system awaiting another?

All this need to be explained properly and added to your README file. In addition to the requirements above.