Amazon Reviews Sarcasm Analysis

- Submit your questions to `eliasof@post.bgu.ac.il`

Getting Started

- Read the assignment description.
- Read the reading the material, and make sure you understand it.
- Write the code that does the sentiment analysis and named entity extraction, 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. …

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 will get as an input text files containing lists of reviews. JSON format. Then, instances will be 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 use-case is as follows:

1. 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*, if received the local application sends a terminate message to the Manager.
2. User gets back an html file containing the reviews with sentiment analysis and sarcasm detection, should lead to the url of the original review (at amazon)

See submission section bellow for instructions of what to submit.

Input File Format

Each input file(there are 5), are written in JSON format. You may use any kit to parse the input, you need not implement it yourself.

Here are the input files:

- 0689835604
- B000EVOSE4
- B001DZTJRQ
- B0047E0EII
- B01LYRCIPG

Output File Format

The output is an HTML file containing a line for each input review, containing the original review colored according to its sentiment:

0 - very negative - dark red 1 - negative - red 2 - neutral - black 3 - positive - light green 4 - very positive - dark green

and followed by [] containing a comma separated list of the named entities found in the review. Following by whether it is sarcastic review or not.

Sarcasm Detection

This is a simple detection algorithm in which we will only check whether the number of stars given by the user are suitable for the review sentiment analysis we applied. If it is so, then there is no sarcasm, but if it is not, then it appears to be sarcasm. Make sure to note that in your output files.

System Architecture

The system is composed of 3 elements:

- Local application
- Manager
- Workers

The elements will 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.

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.
- Downloads the summary file from S3, and create an html file representing the results.
- Sends a termination message to the Manager if it was supplied as one of its input arguments.

IMPORTANT: There can be more than one than one local application running at the same time, and requesting service from the manager.

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:

- If the message is that of a new task it:
  - Downloads the input file from S3.
  - Distributes the operations to be performed on the reviews to the workers using SQS queue/s.
  - Checks the SQS message count and starts Worker processes (nodes) accordingly.
    - The manager should create a worker for every *n* messages, 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.
    - 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*.

- After the manger receives response messages from the workers on all the files on an input file, then 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.

- If the message is a termination message, then the manager:
  - Does 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. Its life cycle is as follows:

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.

The 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, but 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
```

or, if you want to terminate the manager:

```
java  -jar yourjar.jar inputFileName1… inputFileNameN outputFileName1… outputFileNameNn terminate
```

The worker should run as follows:

```
java -cp .:yourjar.jar:stanford-corenlp-3.3.0.jar:stanford-corenlp-3.3.0-models.jar:ejml-0.23.jar:jollyday-0.4.7.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.
inputFileNameI is the name of the input file I.

outputFileName is the name of the output file.
n is: workers - files ratio (how many reviews per worker).

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 *terminate* as one of its arguments.

Technical Stuff

Which AMI Image to Use?

You can choose whatever image you want. You probably want to have one which supports user-data. If you don't want to choose on your own, you can just use this one: ami-b66ed3de use the small one.

The AWS SDK

The assignment will be written in Java, you'll need the SDK for Java for it. We advise you to read the Getting Started guide, and get comfortable with the SDK.

You may use 3rd party Java clients for AWS, such as this or this .

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 for it to do something useful. We refer to this as "bootstrapping" the machine.

The bootstrapping process should download .jar files from an S3 bucket and run them. In order to do that, you need a way to pass instructions to a new node. You can do that using this guide , 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. Here's a code example of how to do that.

If you use an AMI without Java or the AWS SDK for Java, you will need to download and install them via the bootstrap script.

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

Example: `wget http://www.cs.bgu.ac.il/~dsps201/Main -O dsp.html` will download the content at http://www.cs.bgu.ac.il/~dsp201/Main and save it to a file named dsp.html. wget man

Installing from the 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 .

shell scripting with bash: 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.

Since the library we are going to use is a bit large we are going to download from the net, ant we won't pack it with the jar file of the code. Your bootstrapping should download the following files using wget:

- The jar file containing your class files only, without the libraries from S3.
- ejml-0.23.jar
- stanford-corenlp-3.3.0.jar
- stanford-corenlp-3.3.0-models.jar
- jollyday-0.4.7.jar

Tips on Writing to Files using Java
Writing Files .

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

Review Analysis

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)

We need the following libraries for this task:

- ejml-0.23.jar
- stanford-corenlp-3.3.0.jar
- stanford-corenlp-3.3.0-models.jar
- jollyday-0.4.7.jar

Download them and add them to your build path, but **do not** add them to your jar file when exporting your project.

Sentiment Analysis

Note: The methods do not have to be static.

Import List:

14 lines ...

```
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:

20 lines ...

```
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:

23 lines ...

```
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 were 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

Use the Submission System to submit a zip file that contains:

- all sources and binaries, without the libraries that you're supposed to download;
- the output of running your system on:
  - 0689835604
  - B000EVOSE4
  - B001DZTJRQ
  - B0047E0EII
  - B01LYRCIPG
- 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. Yes including 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 of this need to be explained properly and added to your README file. In addition to the requirements above.