SBD Lab Assignment 2

T.C. Leliveld

Contents

1 Introduction

In the previous assignments, you have become familiarized with a number of big data frameworks such as Hadoop and Apache Spark, but we haven't done much supercomputing, let alone on big data. In the second lab of Supercomputing with Big Data we will analyze the Common Crawl, a monthly open-source crawl of the internet. We will be building a lookup table for Dutch phone numbers

and the sites they are referenced in. As one might expect, the Common Crawl is a rather large data set — the July 2017 crawl is 240 TiB uncompressed. To facilitate this analysis, we will make use of Amazon Web Services (AWS). This assignment is based on a Yelp Engineering blog post called, Analyzing the Web For the Price of a Sandwich, and the commoncrawl/cc-pyspark examples.

This is the first year the lab is being held, and as such may experience some teething problems. Feedback is appreciated! The lab files will be hosted on GitHub. Feel free to make issues and/or pull requests to suggest or implement improvements.

2 Goal of this Lab

The goal of this lab is to:

- get hands-on experience with cloud based systems,
- learn about the existing infrastructure for big data and the difficulties with these, and
- learn how to characterize your computation and what machines best fit this profile.

You will work in groups of two. In this lab manual we will introduce you with a big data pipeline for mining dutch phone numbers from a web crawl. You will have to optimize this application according to a metric you deem important. This can be in terms of performance, cost, different analysis or a combination of these. You get the freedom to optimize this application as you see fit.

This lab will be graded on the basis of your report. In the spirit of giving you freedom to find interesting things to optimize, you will not be graded on the achieved result, but on the quality of your analysis and the originality of your contribution. After the reports are handed in, each group will present their suggested improvements and the results they achieved. Each student will also have a discussion with the TAs about their work.

For each optimization you perform, you need to provide a hypothesis why you think this will improve some metric. Try and quantify this as well, giving you some expected result. Report how you implemented the suggested improvement, and finally measure the improvement in the system. Did this match your hypothesis? More interestingly, if it did not, why? Let's illustrate this with an example:

> The analysis at hand takes about 4 hours to run on a cluster of 20 machines. We are interested in optimizing the performance per dollar spent metric. Consider the amount of IO that happens at the start of the computation. The computation consists of analyzing 8TiB of data, thus each machine goes through about 400GB of data. The machines provisioned have a 400 Mbps connection. Each machine spends about 133 minutes downloading. For an extra $0.10 I can upgrade the machines one tier, doubling the connection speed. This moves the price from $0.20 to $0.30 per machine per instance hour.

The machines spend half the time downloading, cutting of an hour of the computation, a 33% increase in speed. The provisioning cost of the machines changes from $20 \cdot 0.20 \cdot 4$, to $20 \cdot 0.30 \cdot 3$, or 16 to 18: a 10% decrease.

This is tested by provisioning m4.xlarge machines instead of m4.large. This resulted in a computation that ran 50 minutes shorter. Due to the baseline being shorter than 4 hours it still resulted in an entire instance hour less than the baseline, so in practice we achieve a slightly smaller performance increase (26%), while still maintaining a 10% decrease in cost. This can be attributed to the CPUs of the machines not being able to keep up with the network connection. This can be seen from the CloudWatch monitoring tools, where it's clear that our CPUs are over utilized, but the machines have leftover network bandwidth.

Finally, please test your hypothesis with classmates, as this will improve everybody's understanding and that is what we are here for after all!

The rest of the document explains how the pipeline works and gives a brief introduction on working with AWS. At the end of the document there's a small section detailing what needs to be done for the report and the presentation.

3 Amazon Web Services

The complete data set we will be looking at is some 8 TiB big, so we need some kind of compute and storage infrastructure to run the pipeline. In this lab we will use Amazon AWS to facilitate this. As a student you are eligible for credits on this platform. There are two options:

AWS normal account Register for a normal account (requires a credit card) and you are eligible for $40 in credits

AWS starter account Register for a educative account (does not require a credit card) and you are eligible for $30 in credits

If possible, we strongly recommend you get a normal account, as there are number of limitations associated with the starter accounts. AWS offers a large amount of different services, but for the first part of this lab only three will be relevant:

EC2 Elastic Compute Cloud allows you to provision a variety of different machines that can be used to run a computation. An overview of the different machines and their use cases can be found on the EC2 website.

EMR Elastic MapReduce is a layer on top of EC2, that allows you to quickly deploy MapReduce-like applications, for instance Apache Spark.

S3 Simple Storage Server is an object based storage system that is easy to interact with from different AWS services.

Note that the Common Crawl is hosted on AWS S3 in the US east region, so any machines interacting with this data set should also be provisioned there.

AWS EC2 offers spot instances, a marketplace for unused machines that you can bid on. These spot instances are often a order of magnitude cheaper than on-demand instances. The current price list can be found in the EC2 website. We recommend using spot instances for the entirety of this lab.

4 Common Crawl

The Common Crawl provides free access to monthly crawl data of the internet. The results are saved in WARC (Web ARChive file format) files. Three different kind of files are hosted by the Common Crawl:

- WARC files which store the raw crawl data,
- WAT files which store computed metadata for the data stored in the WARC, and
- WET files which store extracted plaintext from the data stored in the WARC.

The exact specification is ISO 28500:2017. Seeing how we are only interested in the plain text — as these contain the phone numbers — we will use the WET files.

5 Apache Spark

For this assignment we will use Python 3 in cooperation with Apache Spark. The reason we're using Python (rather than Scala or Java) is the availability of WARC parsing libraries in Python. Every student is free to implement this in his/her language of choice, but the examples will given in Python. Installing Apache Spark (and PySpark) should be straightforward on any Unix based system by using your system's package manager (apt-get, yum, pacman, brew, etc.). We will be using Python 3, which might not be the default Python Apache Spark uses. This can be verified by running PySpark. Ensure that the Spark executables are in your path and run the following:

```
~ pyspark
Python 2.7.13 (default, Jun  5 2017, 14:24:39)
[GCC 4.2.1 Compatible Apple LLVM 8.1.0 (clang-802.0.42)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use ...
17/08/21 14:48:34 WARN NativeCodeLoader: Unable to load native-hadoop ...
17/08/21 14:48:39 WARN ObjectStore: Failed to get database global_temp ...
Welcome to
      ____              __
     / __/__  ___ _____/ /__
```

```
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0
      /_/

Using Python version 2.7.13 (default, Jun  5 2017 14:24:39)
SparkSession available as 'spark'.
```

It is clear that Spark is using Python 2 instead of Python 3 in the above code listing. This behaviour should be changed by setting an environmental variable.

```
export PYSPARK_PYTHON=python3
```

You can also change in which interactive shell PySpark runs. For example, to use the `ipython` shell the following environmental variable needs to be set.

```
export PYSPARK_DRIVER_PYTHON=ipython3
```

In both cases ensure that both `python3` and `ipython3` are in your `PATH`, or insert the complete path in the previous two exports.

For the pipeline, we will use a couple of libraries. Another bash script that downloads and installs these dependencies[1] will be of great use later on. Depending on your platform, you will need to modify the install command. Linux users might need to add `sudo`. Some platforms require a specific version of Python, e.g. `pip-3.4`.

```
#!/usr/bin/env bash

INSTALL_COMMAND="pip3 install"
dependencies="warcio requests requests_file boto3 botocore py4j spark"

for dep in $dependencies; do
    $INSTALL_COMMAND $dep
done;
```

A complete version of this script can be found in the lab's GitHub repository.

6 Building the pipeline

In this chapter, we will talk about the different stages of the standard Big Data Pipeline and how they apply to the analysis we are trying to perform. In the next chapter we will demonstrate how to chain these different stages together.

[1] An experienced Python developer will wonder why we are not using the more idiomatic approach using a `requirements.txt` together with `pip`. This is due to EC2's bootstrapping mechanism being somewhat more straightforward to use with a simple Bash script.

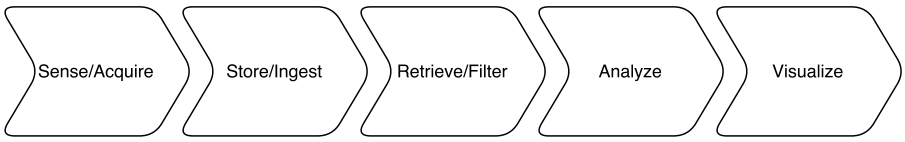

Figure 1: Standard Big Data pipeline.

Sense and Store

As this information is stored on Amazon S3, the sense and store stage are performed by the Common Crawl community.

Retrieve

For the retrieval of data we will write a small bash script that will generate an index of the URLs of the Common Crawl. Additionally we will add functionality that let's extract a few sample files to use for local development.

First off, let's define which crawl we are looking at, where it is hosted, the number of example segments we want to use, and the type of file we are interested in.

```
CRAWL=CC-MAIN-2017-13
BASE_URL=https://commoncrawl.s3.amazonaws.com
LOCAL_SEGMENTS=4
FILE_TYPE=wet
```

The Common Crawl is not hosted as a single 8 TiB file, but rather in small segments, each about 150 MiB in size. Each of these segments has a unique URL. We'd like to download a couple of these segments for local development. Furthermore we will generate two more index files showing where these example files are located locally and on the S3 servers. We will download the location of all the segments, select a couple of files for local development, and download these.

```
local_file_index=input/test_${FILE_TYPE}.txt
s3_sample_file_index=input/test_s3_${FILE_TYPE}.txt

test -d input || mkdir input
test -e $local_files || rm $local_file_index
test -e $s3_sample_file_index || rm $s3_sample_file_index

listing=crawl-data/$CRAWL/$FILE_TYPE.paths.gz
mkdir -p crawl-data/$CRAWL/
wget --timestamping $BASE_URL/$listing -O $listing
gzip -dc $listing | sed 's@^@s3://commoncrawl/@' \
    >input/all_${FILE_TYPE}_$CRAWL.txt
```

```
for segment in $(gzip -dc $listing | head -$LOCAL_SEGMENTS ); do
    mkdir -p $(dirname $segment)
    wget --timestamping $BASE_URL/$segment -O $segment
    echo file:$PWD/$segment >> $local_file_index
    echo s3://commoncrawl/$segment >> $s3_sample_file_index
done
```

A complete version of this script can be found in the lab's GitHub repository.

Filter

To identify phone numbers, we need some way to filter these from plaintext. The Dutch phone number format can be written in two ways, the international and national form. To keep the filtering relatively simple, we will limit ourselves to the international form, as the national form is only identified with a single zero prefix, causing false positives as any 10 digit number starting with a 0 will now be identified as a phone number.

The international format starts with either a +31, or 0031, then an optional "(0)", and finally 9 digits. We allow a variety of marks to be inserted between the numbers: spaces, parenthesis, and dashes. The following regular expression captures the desired behaviour.

```
(?:(?<=\D)00[\(\)\- ]*3[\(\)\- ]*1|\+[\(\)\- ]*3[\(\)\- ]*1)
(?:[\(\)\- ]*\( *?0 *?\))?(?:[\(\)\- ]*[0-9]){9}
```

We would like to convert all matched phone numbers to a single format, so duplicates can be matched. First we remove the optional parenthesised zero, and then we remove any additional punctuation. The following regular expression can be used to remove these.

```
(?:[\(\)\- ]*\( *?0 *?\))|[\(\)\- ]*
```

Finally, we convert all the following numbers to the "+31" format. Effectively this means we need to replace all the "0031" prefixes to "+31". The following regular expression can be used to substitute "+31".

```
^00
```

In Python we can compile these regular expressions for faster repeated usage.

```
import re

class PhoneFilter:
    phone_nl_filter = re.compile(phone_regex)
    clean_filter = re.compile(replace_regex)
    zero_to_plus_filter = re.compile(zeroplus_regex)
```

```
    def find_phone_numbers(self, content):
        numbers = self.phone_nl_filter.findall(content)
        num_filt = {re.sub(self.zero_to_plus_filter, "+",
                    re.sub(self.clean_filter, "", num)) for num in numbers}
        for n in num_filt:
            yield n
```

Analysis

We would like to analyze the phone numbers and the websites they are referenced on, after we complete the filtering stage. To allow for this analysis, we want to store the results in a suitable format. Apache Spark has introduced DataFrames, from the documentation:

> A Dataset is a distributed collection of data. Dataset is a new interface added in Spark 1.6 that provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL's optimized execution engine.
>
> A DataFrame is a Dataset organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood.

This is a common pattern, where we use RDDs to go from unstructured data, and filter it and organize it so that we end up with (semi-)structured data. This data can than be more effectively analyzed using the Spark SQL tools, among which DataFrames.

To use DataFrames we need to specify a schema. In our case the schema will be the following.

```
from pyspark.sql.types import StructType, StructField, StringType, ArrayType

output_schema = StructType([
    StructField("num", StringType(), True),
    StructField("urls", ArrayType(StringType()), True)
    ])
```

Visualization

We do not have anything planned here, but if someone knows a cool way to visualize this data, let us know!

7 Chaining the Pipeline Together

In this chapter we will demonstrate how we can use Apache Spark to chain this pipeline together. First off we need to load the relevant input segment index.

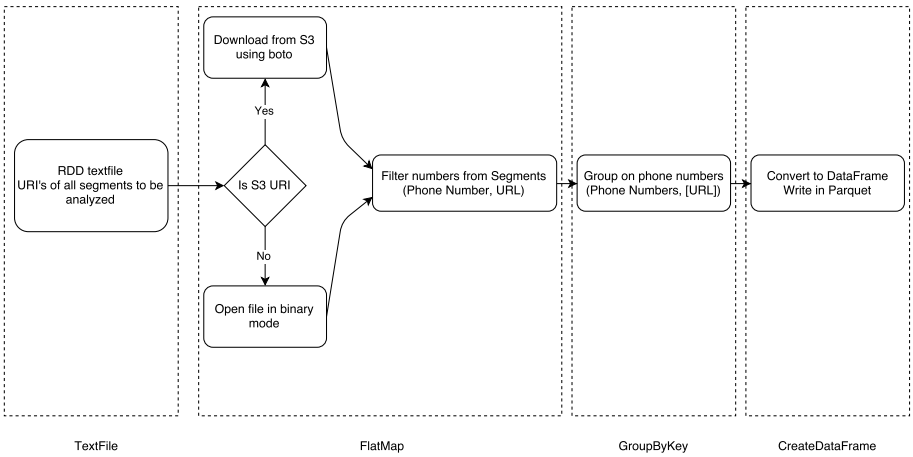

Figure 2: Overview of the analysis and different RDD's.

```
import spark
import spark.sql

class PhoneNumberExtractor:

    def __init__(self, input_file, output_dir, name, partitions=None):
        self.name = name
        self.input_file = input_file
        self.output_dir = output_dir
        self.partitions = partitions

    def run(self):
        sc = SparkContext(appName=self.name)
        sqlc = SQLContext(sparkContext=sc)

        self.failed_record_parse = sc.accumulator(0)
        self.failed_segment = sc.accumulator(0)

    def process_segments(self, segment_uri):
        stream = None
        if segment_uri.startswith('file:'):
            stream = self.process_file_warc(segment_uri)
        elif segment_uri.startswith('s3:/'):
            stream = self.process_s3_warc(segment_uri)
        if stream is None:
            return []
        return self.process_records(stream)
```

If the URIs from the input file refer to a local file, we can simply return the file (in binary mode).

```
def get_file(self, file_uri):
    return open(file_uri, 'rb') # Binary mode
```

If the URI refers to an S3 link, we need to download it first. We use `boto3`, a library for interacting with S3, to download the segments. To download a segment, a file handle has to be supplied. We can use `TemporaryFile` to generate temporary files (and handles). These files are automatically removed when the file handle is garbage collected.

```
def get_s3(self, s3_uri)
    no_sign_request = botocore.client.Config(signature_version=botocore.UNSIGNED)
    s3client = boto3.client('s3', config=no_sign_request)
    s3pattern = re.compile('^s3://([^/]+)/(.+)')
    s3match = s3pattern.match(uri)
    bucketname = s3match.group(1)
    path = s3match.group(2)
    warctemp = TemporaryFile(mode='w+b')
    s3client.download_fileobj(bucketname, path, warctemp)
    warctemp.seek(0)
    return warctemp
```

We are left with decoding the input stream into different records, filtering the phone numbers, and yielding (number, URL) pairs.

```
def process_records(self, stream):
    for rec in ArchiveIterator(stream):
        uri = rec.rec_headers.get_header("WARC-Target-URI")
        if uri is None:
            continue
        content = rec.content_stream().read().decode(utf-8)
        for num in self.find_phone_numbers(content):
            yield (num, uri)
```

Finally we write away these results to the output destination.

```
def run(self):
    sc = SparkContext(appName=self.name)
    sqlc = SQLContext(sparkContext=sc)

    if self.partitions is None:
        self.partitions = sc.defaultParalellism

    self.failed_record_parse = sc.accumulator(0)
    self.failed_segment = sc.accumulator(0)

    index = sc.textFile(self.input, minPartitions=self.partitions)
    phone_numbers = index.flatMap(self.process_warcs)
    phone_numbers_grouped = phone_numbers.groupByKey().mapValues(list)
```

```
    sqlc.createDataFrame(phone_numbers_grouped, schema=self.output_schema) \
            .write \
            .format("parquet") \
            .save(self.output)
```

Additionally we provide an argument parser that allows us to dynamically select different input files, output files, name of the application, and the number of partitions in the input RDD.

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser("Phone number analysis using Apache
    Spark")
    parser.add_argument("--input", '-i', metavar="segment_index",
                        type=str, required=True,
                        help="uri to input segment index")
    parser.add_argument("--output", '-o', metavar="output_dir",
                        type=str, required=True,
                        help="uri to output directory")
    parser.add_argument("--partitions", '-p', metavar="no_partitions",
                        type=int,
                        help="number of partitions in the input RDD")
    parser.add_argument("--name", '-n', metavar="application_name",
                        type=str, default="Phone Numbers",
                        help="override name of application")
    conf = parser.parse_args()
    pn = PhoneNumbers(conf.input, conf.output,
                      conf.name, partitions=conf.partitions)
    pn.run()
```

The advantage of this is that we can quickly try our program against different segment indices. This determines whether we will download the files from AWS, or use the local files, and how many segments we will analyze (i.e. the number of segment URIs in the input segment index).

A complete version of this program can be found on the lab's GitHub repository. Play around with the script, and reading the data. How many phone numbers can you find in the four sample files? Try and estimate how many there will be in the complete data set based on this.

8 Using AWS

We will be using the AWS infrastructure to run the pipeline. I assume everyone has an AWS account at this point. Log in to the AWS console, and open the S3 interface. Create a bucket where we can store the scripts, the segments index files, and the output from the pipeline.

There are (at least) two ways to transfer files to S3:

1. The web interface, and

2. The command line interface.

The web interface is straightforward to use. To use the command line interface, first install the AWS CLI.

To copy a file

```
aws s3 cp path/to/file s3://destination-bucket/path/to/file
```

To copy a directory recursively

```
aws 3 cp --recursive s3://origin-bucket/path/to/file
```

To move a file

```
aws 3 mv path/to/file s3://destination-bucket/path/to/file
```

The aws-cli contains much more functionality, which can be found on the AWS-CLI docs.

Move the phone analyzer script, the dependency shell script, and the segment indices to your S3 bucket.

We are now ready to provision a cluster. Go to the EMR service, and select *Create Cluster*. Next select *Go to advanced options*, select the latest release, and check the frameworks you want to use (in this case Spark, and Hadoop). We need to enter some software settings specifically to ensure we are using Python 3. Enter the following in the *Edit software settings* dialog. A copy paste friendly example can be found on the AWS site.

```
[
    {
    "Classification": "spark-env",
    "Configurations": [
            {
                "Classification": "export",
                "Properties": {
                    "PYSPARK_PYTHON": "/usr/bin/python3"
                }
            }
        ]
    }
]
```

EMR works with steps, which can be thought of as a job, or the execution of a single program. You can choose to add steps in the creation of the cluster, but this can also be done at a later time. Press *next*.

In the *Hardware Configuration* screen, we can configure the arrangement and selection of the machines. I would suggest starting out with *m4.large* machines on spot pricing. You should be fine running an example workload with a single master node and two core nodes.[2] Be sure to select *spot pricing* and place an appropriate bid. Remember that you can always check the current prices in the information popup or on the Amazon website. After selecting the machines, press *next*.

In the *General Options* you can select a cluster name. You can tune where the system logs and a number of other features (more information in the popups). To install the dependencies on the system you need to add a *Bootstrap Action*. Select *Custom action*, then *Configure and add*. In this pop-up, give this action a appropriate name, the *Script location* should point to the dependency shell script in your previously created S3 bucket. Be aware that the correct install command on EC2 instances is `sudo pip-3.4`. You can leave the *Optional arguments* dialog empty. After finishing this step, press *next*.

You should now arrive in the *Security Options* screen. If you have not created a *EC2 keypair*, I highly recommend that you do so now. This will allow you to forward the Yarn and Spark web interfaces to your browser. This makes debugging and monitoring the execution of your Spark Job much more manageable. To create a *EC2 keypair*, follow the AWS instructions.

After this has all been completed you are ready to spin up your first cluster by pressing *Create cluster*. Once the cluster has been created, AWS will start provisioning machines. This should take about 10 minutes. In the meantime you can add a step. Go the *Steps* foldout, and select *Spark application* for *Step Type*. Clicking on *Configure* will open a dialogue in which you can select the application location in your S3 bucket, as well as provide a number of argument to the program, spark-submit, as well as your action on failure. We need to provide a number of arguments to the program to specify our input segment index, and output directory at the very least. An example argument list is included below

```
-i s3://sbd-ex/input/test_s3_wet.txt
-o s3://sbd-ex/output
-p 4
-n "example arguments"
```

Before the setup has finished you should also configure a proxy for the web interfaces. More detailed information can be found on the AWS website. You can check the logs in your S3 bucket, or the web interfaces to track the progress of your application and whether any errors have occurred. If everything went well, you should have output in your S3 bucket. Check this on your machine by copying the files and inspecting the results in a Spark shell by loading it via SQLContext.

[2] By default, there are some limitations on the number of spot instances your account is allowed to provision. If you don' t have access to enough spot instances, the procedure to request additional can be found in the AWS documentation.

```
~ pyspark
Python 3.5.1 (default, Mar  6 2016, 15:01:56)
Type "copyright", "credits" or "license" for more information.

IPython 4.0.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra...
Using Spark's default log4j profile: org/apache/spark/log4j-d...
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For Spa...
17/08/31 17:01:56 WARN NativeCodeLoader: Unable to load nativ...
17/08/31 17:02:01 WARN ObjectStore: Failed to get database gl...
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0
      /_/

Using Python version 3.5.1 (default, Mar  6 2016 15:01:56)
SparkSession available as 'spark'.

In [1]: sqlc = SQLContext(sc)

In [2]: df = sqlc.read.parquet("./output")

In [4]: df.count()
Out[4]: 106
```

9 Report

We have become familiar with both the pipeline in this exercise, as well as the AWS infrastructure. Now determine a metric you are trying to optimize. Based on this metric find areas that you could improve the pipeline in. Remember to report the following:

- hypothesis (preferably quantitatively) about what your change is going to do,
- implementation, how did you go about implementing the change, and
- results, was your hypothesis correct (if not, why?).

As a starting point you can make a good analysis of the application. Figure out what kind of of I/O is happening (e.g. how much MB's does each machine have to download) versus the amount of compute time that is happening. You can try and find the optimal machine for your metric. You can compare this to a "general purpose" m4.large machine.

A word of advice: Be careful not to overspend your credits! It is your responsibility to ensure you are not blowing all your credits straight away. How much data do you need to process to get a reasonable indication of the performance on the entire dataset? Can you make meaningful prediction from working with smaller sample sizes? Try and extrapolate this to the entire dataset.

10 Presentation

Each group will have a short presentation where they present their proposed improvements, and how they panned out. This is not graded, but rather an informal session to see what your fellow students tried, and how that worked out. Prepare 1-2 slides per improvement you implemented, containing the aforementioned three points. The date and location of this session will be announced later.