ENEE 657 – Security Analytics Homework

Homework Due: 20 September 2017 at 11:59 pm.

Submission Instructions: Write one Python program, following the instructions below. Submit it from the GRACE machines, along with the corresponding output files, using the following commands:

```
submit 2017 fall ENEE 657 0101 2 spark_problem.zip
```

1 Homework overview

The learning objective of this homework is for students to gain first-hand experience with some data analytics techniques that are commonly used to solve security problems. *Locality sensitive hashing* (LSH) is a technique for approximate clustering and nearest-neighbor search. For example, locality sensitive hashing may be applied to streaming Twitter posts to identify posts that are similar to a corpus of documents containing exploit code. The *Spark* data analytics platform allows you to perform some of these operations efficiently at scale.

2 Initial setup

Use the new Ubuntu VM available for Homework 2. The VM includes all the tools you need for this homework. You need to download the upgraded VM again, you can find it here:

```
http://www.umiacs.umd.edu/~tdumitra/courses/ENEE657/Fall17/homeworks.html
```

This is the machine we will use for testing your submissions. It has $openjdk - 7$ and $spark - 1.4.0$ installed. If your submission doesn't work on that machine, you will get no points. It makes no difference if your submission works on another Ubuntu version (or another OS).

You can find links to several tutorials at the address above; read these documents if youre stuck. I also encourage you to ask questions on our Piazza message board.

Starter files. Starter files are available on the class web page:

```
http://www.umiacs.umd.edu/~tdumitra/courses/ENEE657/Fall17/homeworks.html
```

3 Task 1: Evaluating document similarity in real time

The materials you need for this task are provided in the starter files, under the `locality_sensitive_hashing` directory.

In this task, you are given a pre-existing model built from a set of documents of interest (e.g. tweets that refer to the ShellShock vulnerability), and you must check, in real time, which of the documents from an incoming stream of data are most similar to the documents of interest. A document D is represented as a set of the words $W_D = \{w_1, w_2, ...\}$ that appear in the document.

Locality Sensitive Hashing. A standard measure of the similarity between two documents D_1 and D_2 is the Jaccard index $J(D_1, D_2) = \frac{|W_{D_1} \cap W_{D_2}|}{|W_{D_1} \cup W_{D_2}|} \in [0, 1]$. The Jaccard index of two documents is 1 if the documents are identical and 0 if they don't have any words in common. However, the set intersection and union needed to compute the Jaccard are expensive operations, which makes it difficult to use this similarity measure when dealing with large data sets.

Instead, it is possible to approximate the Jaccard index with *MinHashing*. Given the set W of all possible words that may appear in the documents, this technique requires a random permutation function $h : W \mapsto \{1, ..., |W|\}$. In other words, $h(w)$ assigns a unique rank to word w; in practice, you can use a collision-resistant hash function for h. The MinHash of a document D is first word from W_D in the ranking given by h: $MinHash_h(D) = \arg\min_{w \in W_d} h(w)$. For two documents D_1 and D_2, the probability of their MinHashes being the same is equal to their Jaccard index $J(D_1, D_2)$.

The MinHash is often not enough to conclude whether two documents are similar. The idea behind *locality sensitive hashing (LSH)* is to compute multiple hashes and map documents into buckets; similar documents are likely to be mapped to the same bucket. With LSH, you need $n = b \times r$ different hash functions and you compute n MinHashes for each document. You then arrange these MinHashes into b bands, each band having r rows. Table **??** illustrates LSH for three documents, D_1, D_2, and D_3. We compute six MinHashes, partitioned in three bands with two rows per band. Each band has its own buckets (clusters). Two documents are in the same bucket if their MinHash values match for all the rows in the band (recall that in this case the MinHash of a document is a word). In our example, the MinHashes for documents D_1 and D_2 match in both Bands 1 and 2, so they are in the same bucket in those bands, while in Band 3 documents D_1 and D_2 are in the same bucket. Two documents are considered similar if they appear together in at least one bucket; the documents have b chances of appearing in the same bucket. The probability that two documents D and D' are considered similar, when using LSH, is $1 - (1 - J(D, D')^r)^b$.

Matching an LSH model. Your task is to write a program to find the candidate subset of documents (tweets that might refer to the ShellShock vulnerability) that match the content of a pre-built model (tweets that are known to talk about ShellShock) using LSH. All operations must be implemented in *Spark*, using parallel collections (*RDDs*). Your program should load documents from disk and distribute them amongst the Spark workers. Each worker is responsible for doing text normalization (removal of multiple white spaces & non-ASCII characters, lowercase conversion, etc.). The workers then compute the LSH for each of the documents and check the content match against a set of documents in the model. The interface returns None for documents with no match; these entries should be filtered out before saving the results to the output folder. The final list

Table 1: Locality Sensitive Hashing Example

	h	D_1	D_2	D_3	Buckets
Band 1	$MinHash_{h_1}$	w_1	w_1	w_1	$[D_1, D_2]\,[D_3]$
	$MinHash_{h_2}$	w_1	w_1	w_2	
Band 2	$MinHash_{h_3}$	w_5	w_5	w_3	$[D_1, D_2]\,[D_3]$
	$MinHash_{h_4}$	w_8	w_8	w_4	
Band 3	$MinHash_{h_5}$	w_1	w_7	w_7	$[D_1]\,[D_2, D_3]$
	$MinHash_{h_6}$	w_6	w_6	w_6	

should contain the ids of the tweets that are considered candidate matches and can be further verified using an exact match algorithm. Your program should save this list to disk.

Starter files. The starter files include a partially completed Python program, located at `/homework/minhash_homework.py`. Complete this program by adding the appropriate code to replace all the `None` values in the `__main__` function. The section where you need to add your code is delimited by comments; you must complete 7 steps, in order (each step depends on the completion of the previous steps). The data files are in the `data` subdirectory. Invoke the program like this:

```
$ cd /home/seed/Documents/spark/bin/
$ ./spark-submit homework/minhash_homework.py > data/solution.out
```

You can also find these files in the VM, under `/home/seed/Documents/spark/bin`. The output of your program will be located in `data/output_folder`. The output folder must be deleted between consecutive runs:

```
$ rm -r data/output_folder/
```

Submitting.

Make sure you un-comment the *print* instructions after completing each step. Note that the output could get quite verbose. The program's *stdout* will be evaluated as part of your grade. Create an archive, called `spark_problem.zip`, which contains the `data\output_folder`, the `data\solution.out` and your completed `minhash_homework.py` script. The starter files include a shell script, called `pack_files.sh`, that will create this archive for you (run `chmod +x pack_files.sh` if you can't execute it). Submit the `spark_problem.zip` file as described at the beginning of this handout.

Extra notes.

This part is not graded. In order to make the task more practical we would need to address some additional concerns.

The task requires you to retrieve the documents that match at least some tweet from the ground truth. The LSH model for this homework is prebuilt. Could you build the model yourself from the model original files?

How would you modify the code in order to retrieve the ids of tweets that were matched in the

ground truth?

The homework runs on the VM and it creates a worker thread that gets to perform matching on the entire input set of tweets. However, for large datasets, we would need to spawn multiple workers, each responsible for a partition of the dataset. How could you create additional workers and distribute the input among them?