Assignment

Anomaly Detection using Negative Selection Algorithms

Deadline March 9th, 2021

Handout for the *Natural Computing* lecture, February 23rd, 2021

Johannes Textor, Inge Wortel

Objectives of This Exercise

1. Familiarize yourself with the concepts behind negative selection.
2. Apply a negative selection algorithm to a real-world dataset.

In this assignment, you will apply the negative selection algorithm to anomaly detection in sequence data. The assignment consists of two parts: a "walk-through" part, in which the algorithm implementation is explained using an artificial toy problem, and the assignment proper, in which you are expected to apply this implementation to a more realistic dataset from the domain of network security.

This assignment description contains some introductory explanation and text. You are not expected to respond to these parts in your report. Respond only to those parts that are labelled as **your task**.

1 Using the Negative Selection Algorithm

For this assignment, you can work with the author's implementation of a negative selection algorithm (although you are of course very welcome to write your own implementation, should you so wish). This implementation is written in Java, which you need to have installed. The program itself and the data files needed for this assignment can be found in the file `negative-selection.zip`, which you can download from Brightspace.

Familiarize yourself with the implementation

Download the file `negative-selection.zip` and unpack it. This will create a folder `negative-selection`. Open a terminal and change to that folder. Run the negative selection program using:

```
java -jar negsel2.jar
```

This will show a list of possible options.

Classifying languages

In this toy example, your task is to train an artificial T cell repertoire using fixed-length strings taken from the English-language book *Moby Dick*. Then, you will expose this repertoire to both English-language strings and strings from another language, which will be the "anomalies". If the algorithm is tuned well, it should be able to distinguish the English-language strings from those of the other language.

We start by training our repertoire on English strings. Run the following command:

```
java -jar negsel2.jar -alphabet file://english.train -self english.train -n 10 -r 4 -c
```

This will build a repertoire containing all patterns of length 10 (switch -n) that do not share any contiguous substring of length more than 4 (switch -l) with any string in the input set `english.train` [1] (where one string per line is given). It will then read input strings line by line from standard input. For each line, it will count (-c)

[1] In the lecture, we called these patterns *r-contiguous detectors*, and here we have $r = 4$.

the number of patterns in the repertoire that match this line. If the -c switch is omitted, it will instead output the length of the longest contiguous string shared by the input line and any pattern in the repertoire. Lastly, the -alphabet switch determines the basis alphabet for strings and patterns to be the set of unique characters found in the file given as the argument. The default behaviour is to use all characters found in the input file, so the switch can be safely omitted in this case. (It will be relevant in the next exercise, however, and is therefore introduced here.)

When the program expects your input, type in the following line:

```
call_me_is
```

This will output the number 0, because this string is the first line from the input dataset. Next input the line

```
fall_me_is
```

As we can see, this string can be matched by a vast number of patterns. Such huge numbers can be unwieldy to work with, so there is an option that lets us output each number x. as $\log_2(1+x)$ instead. Test this feature by ending the current session (by typing CTRL-D to end the input or CTRL-C to abort the program) and restart the program with the -l switch:

```
java -jar negsel2.jar -self english.train -n 10 -r 4 -c -l
```

Now, the output for `call_me_is` is still 0, but the output for `fall_me_is` is about 28.

Using input redirection, we can now determine the number of matching patterns for each string in the files `english.test` and `tagalog.test`. For example, on Linux, Mac or cygwin platforms, we could use a small inline `awk` script to compute the average number of matching patterns for both files:

```
java -jar negsel2.jar -self english.train -n 10 -r 4 -c -l < english.test | awk
    '{n+=$1}END{print n/NR}'
java -jar negsel2.jar -self english.train -n 10 -r 4 -c -l < tagalog.test | awk
    '{n+=$1}END{print n/NR}'
```

As we can see here, the average number of matching patterns is somewhat less than 2-fold higher for Tagalog strings than for English strings, so some learning does seem to occur.

Your task

1. Compute the area under the receiver operating characteristic curve (AUC) to quantify how well the negative selection algorithm with parameters $n = 10, r = 4$ discriminates individual English strings from Tagalog strings by using the files `english.train` for training and `english.test` as well as `tagalog.test` for testing. [2]
2. How does the AUC change when you modify the parameter r? Specifically, what behaviour do you observe at $r = 1$ and $r = 9$ and how can you explain this behaviour? Which value of r leads to the best discrimination?
3. The folder `lang` contains strings from 4 other languages. Determine which of these languages can be best discriminated from English using the negative selection algorithm, and for which of the languages this is most difficult. Can you explain your findings?

[2] You compute the AUC curve by merging the two test string sets, and then sorting them by the "anomaly score" that is given to them by the negative selection algorithm. Then, for each possible cut-off score (that is, each distinct value in this list), you compute the sensitivity (percentage of "anomalous" strings higher than that score) and the specificity (percentage of "normal" strings lower than that score). Then you generate the AUC curve from those values. Or, simply use an R package such as AUC.

2 Intrusion Detection for Unix Processes

We will now apply negative selection to a more interesting problem: intrusion detection on a Unix computer system.

Unix processes communicate with the kernel using system calls (such as `open`, `read`, `write` etc). Tools such as `strace` can be used to monitor the system calls made by a running process. Anomalous system call sequences (such as a normally silent process suddenly opening hundreds of files) could indicate an exploit.

In this exercise, we will work with such data, which is stored in the folder `syscalls` and has been collected many years ago in the group of Stephanie Forrest at the University of New Mexico[3]. This data has a similar structure to the language data from the previous exercise:

- Files ending with `.train` contain training sequences. All these are system call sequences recorded during normal behaviour of a process such as `sendmail`.
- Files ending with `.test` contain system call sequences representing both normal and anomalous behaviour.
- Each `.test` file has a corresponding `.labels` file that contains a 0 for a normal sequence and a 1 for an anomalous sequence.
- Finally, the `.alpha` file contains the alphabet used to encode the system calls – each symbol represents a different system call.

There are two subfolders in the folder `syscalls`, each containing the aforementioned files.

Your task: use negative selection to detect the anomalous sequences in the system calls datasets! Perform an AUC analysis to evaluate the quality of your classification.

There are two important differences to the "toy example" above.

1. The data format differs slightly, with the classification being stored in the separate `.labels` rather than having two different files for normal and anomalous data.
2. More importantly, the sequences stored in the files are no longer of a fixed length. For training, this means that you will need to pre-process each sequence to a set of fixed-length chunks (for instance, you could use all substrings of a fixed length, or all non-overlapping substrings of a fixed length). For classification, you also need to split the sequences into chunks, compute the number of matching patterns for each chunk separately, and merge these counts together to a composite anomaly score (for instance, you could average the individual counts).

Choose the parameters n and r for the negative selection algorithm yourself. You can use the parameters from the language example as a starting point.

[3] http://www.cs.unm.edu/~immsec/systemcalls.htm