CS 5540 Project 1

Arjun Biddanda (aab227) and Muhammad Khan (mhk98)

February 18, 2013

1 Determining Neuron Sensitivity

1.1 Stimulus Class

The `Stimulus` class is used to give an object definition in Java for each particular stimulus. We define the stimulus as a Java `enum` (a strict or-type enumerative declaration), so there are no fields that we have to extend for each stimulus. The stimuli are used as defining pieces of data with regards to our construction of the `Neuron` class.

1.2 Neuron Class

This is a class that we developed in order to accurately represent a neuron. The defining feature of the class is the map that is representative of the spike trains that are recorded for each stimulus. We define the map as *spiketimes*, which is a hashmap which takes a stimulus as a key and outputs a double array detailing all of the spike times for that particular stimulus.

1.2.1 Sensitive Stimuli

This is a simple method to compute (roughly) whether a neuron is sensitive to a a particular stimulus. We first create an outer loop to iterate through the list of all possible stimuli, where we choose a stimulus s. Within that outer loop, we have two inner loops to iterate through the different spike time trials for s. We have an average counter avg which maintains the average spike number from all of the trials that we have gone through for that particular stimulus.

At the termination of the inner loop we have a check to decide if the average number of spikes is high enough for us to determine that it is significant. We have set the "cutoff" point for the number of spikes to be an average of 30 for a given

stimulus s. This was arbitrarily decided from looking at the training data that we had divided our set into.

After applying this to the set of all neurons presented to us, we would theoretically have a length 35 array of variable lists containing the significant stimuli to the neuron at whichever index i we were at along the array.

1.3 DataParser

In order to use the data provided within the text file, we had to develop a parser which will create all of the neurons that we require. The parser is not optimized for speed in any way, because it just sequentially goes through the text file and every time that it hits a line that begins with the character 'N' it just remembers to create a new Neuron object and add the old one to the accumulator list. At the termination of the algorithm we will have an array filled with 35 neuron objects (because that is how large our data set is).

2 Creating a Response Space

2.1 Initial Assumptions

Our initial method by which to calculate significant stimuli was quite naive really. Since we are going purely by the number of spikes that are generated, the spread of the data is not considered whatsoever, and that is very dangerousin our interpretation of this data. For instance, if we are given spike trains A and B, where A has a large number of spikes at the ends of the time intervals, and B has an equally large number of spikes in the middle. According to our first model, they would appropriately be assigned to the fact that they have the same significant stimuli.

This is clearly not the case, because we can imagine that visually these spike trains do not look alike at all. So within our revised model to create the Response Space, we included a more substantiated approach in order to account for the spread of the data.

- We must first isolate each Neuron-stimulus pair and then calculate "how close" these responses are.
- In order to create this mapping we developed a matching algorithm which calculates how much it "costs" to transform spike trains into on another.
- plotting all of these costs, we can determine distances between all the different spike trains.

- We then apply a k-Nearest Neighbors algorithm in order to determine if the stimulus is significant for that particular neuron.

2.2 ResponseSpaceCalculator

The initial methods present within this class are from an old idea of calculating the correlation coefficient between two sets of floating point numbers, but that was dropped because we cannot guarantee that the length of the two sets will be equal.

2.2.1 Align

This method relies on the core assumption that A and B are of equal length, where A and B are sets of floating point numbers. We set the initial cost of transformation (*cost*) to 0. We then iterate through A and then check if it is equal to B at the index i. If it is not, then we must transform it with cost ΔT which is then added onto *cost*. At the end we return *cost*.

2.2.2 Align2

Here we eliminate the initial assumption on equal length that is present within Align. We can eliminate this with two simple cases:[1]

- $length(A) < length(B)$, then we add more elements to A in order to match the length of B
- The vice-versa of the above statement, we subtract from A in order to match the length of B

[1]Note: This part of the algorithm was developed rather roughly, and a better optimal edit-distance function would have been made if there were more time