MountainSort User Guide

Jeremy F. Magland
Center for Computational Biology, Simons Foundation

January 5, 2016

Abstract

This is an overview of the MountainSort software for automated spike sorting.

1 Introduction

Our objectives are threefold. First we will implement an automated spike sorting algorithm requiring minimal user intervention. Second, we will provide interactive visualization tools for exploring the output of this and other spike sorting softwares. Finally, we will provide an algorithm-independent framework for objectively validating spike sorting performance by reporting a reliability metric for each detected neuron.

2 Sorting automation

Manual human intervention as part of a spike sorting pipeline is problematic for a number of reasons. The workload may not be practical for a single operator since hundreds of large datasets can be acquired over the course of a few days. Operator bias becomes an issue if the task is shared among several users or the sorting methods of a single user drift over time. Assessing reproducibility or sorting reliability is near impossible since the user cannot be asked to supervise processing of the same dataset multiple times in an unbiased manner. Finally, human-supervised processing cannot easily be transferred between labs and is not combatible with objective comparisons between alternative methods.

With that said, it is not reasonable to expect a single algorithm to work for all datasets. The experimental setup, noise properties, and hardware-specific artifacts vary significantly between laboratories. A certain degree of tweaking of parameters should be expected. Whenever possible, these should be applied during the preprocessing stage rather than at the end of automated sorting.

3 The algorithm

3.1 Preprocessing

The input to the preprocessing stage is an $M \times N$ array Y of raw data collected over N timepoints and M electrode channels. The output is two $M \times N$ preprocessed arrays, $Y_{\texttt{filt}}$ and $Y_{\texttt{white}}$. The first is used for visualization only and the second is used in the subsequent sorting steps. The options are documented in the MATLAB code.

```
Algorithm 1: PREPROCESS
    Input: $Y$ - $M \times N$ array of raw data
    Input: $o_{\texttt{filter}}$, $o_{\texttt{whiten}}$, $o_{\texttt{filter2}}$ - options
    Output: $Y_{\texttt{filt}}$ - $M \times N$ array of bandpass filtered data
    Output: $Y_{\texttt{white}}$ - $M \times N$ array of bandpass filtered and whitened data
  1 $Y_{\texttt{filt}} \leftarrow \texttt{bandpass}(Y, o_{\texttt{filter}})$
  2 $Y_0 \leftarrow \texttt{whiten}(Y_{\texttt{filt}}, o_{\texttt{whiten}})$
  3 $Y_{\texttt{white}} \leftarrow \texttt{bandpass}(Y_0, o_{\texttt{filter2}})$
```

3.2 Detection

Events are detected on a per-channel basis. The output is a collection of timepoints (for each channel) where spike events have been detected. The same event will usually be identified on more than one electrode channel, but this redundancy is handled in subsequent steps. Events are identified whenever the absolute voltage exceeds $\tau \cdot \sigma$ where τ is a threshold parameter (e.g., $\tau = 5$) and σ is the standard deviation of the signal on a particular channel. However, to avoid detecting the same event more than once, we do not allow two events to occur within a certain number of timepoints W (for example 40). Specifically, an event occurs at timepoint t and channel m if

$$|Y(m,t)| \geq \tau\sigma$$

and

$$|Y(m,t)| > |Y(m,t')| \; \forall t' \text{ s.t. } |t-t'| \leq W.$$

3.3 Feature Extraction

Principal component features are extracted for each channel separately, using data from the channel and its neighbors. The set of neighboring channels is determined by the geometric electrode layout and is specified by an $M \times M$ adjacency matrix. Time clips of T timepoints centered around each event are extracted, and principal component analysis is applied on resulting set of vectors. The dimension of the vectors is $M_m \cdot T$ where M_m is the number of channels adjacent to channel m. The output for each channel is a $P \times E_m$ array of PCA features, where P is the number PCA components (e.g., 3) and E_m is the number of events detected on channel m.

3.4 Clustering

Clustering is performed on each channel separately by applying the ISO-SPLIT algorithm to the extracted feature vectors. This algorithm is designed to have zero adjustable parameters. For example, unlike k-means, the number of clusters does not need to be specified in advance. It is a density-based algorithm that assumes that clusters are unimodal and that any two distinct clusters are separated by a region of relatively lesser density. A statistical test is used at each iteration to determine whether clusters should be merged or redistributed. More details can be found on the arXiv (http://arxiv.org/abs/1508.04841).

The output for each channel m is a list of integer labels between 1 and K_m, one for each event, where K_m is the number of clusters identified by the algorithm.

3.5 Cluster Splitting

The clustering stage above only uses data from the neighborhood of each electrode channel. Sometimes this is insufficient to separate clusters that may differ only on channels that are geometrically separated from the primary loading channel. Therefore a second clustering stage is used to test each previously-identified cluster for splitting, using data from all channels. Again, PCA features are extracted and ISO-SPLIT is employed.

3.6 Template Extraction

A $M \times T$ template W_k is computed for each cluster $k = 1, \dots, K$ identified in the previous steps. This is done by taking the average of time clips of size T extracted at all event timepoints labeled as cluster k. Note that at this stage the set of templates will have redundancies since the same neuron will usually be identified on more than on channel.

3.7 Consolidation

As mentioned above, the same neuron will usually be identified on more than one channel, and therefore redundancies need to be eliminated. In most cases, each neuron will have a primary electrode on which it loads with greater energy than on any other electrode. Therefore, if cluster k was identified from channel m_k and

$$\|W_k(m_k, \cdot)\|_2 < \mu \|W_k(m', \cdot)\|_2$$

for some other channel m', then we know that k is a redundant cluster and can therefore be eliminated. Here, $\mu < 1$ is a tolerance constant (e.g., $\mu = 0.9$). In other words, we eliminate all clusters that whose primary load channel is not the channel used to identify the cluster.

The above heuristic handles most redundancies. However, sometimes a neuron fires with around the same energy on two nearby electrodes. We therefore use a second criterion (in a second pass) to eliminate these situations. We decide that two clusters k_1 and k_2 on different channels are the same if

$$\|W_{k_1} - W_{k_2}\|_2 \leq \eta W_{k_i}$$

for both $i = 1$ and $i = 2$, where $0 < \eta < 1$ is a constant (e.g., $\eta = 0.5$). Note that all time shifts of W_{k_1} must be considered since the peak amplitude on channel m_{k_1} will often occur at a slightly different time than on channel m_{k_2}. In the case of a duplicate, we remove the cluster corresponding to the smaller (in L_2 norm) template.

3.8 Fitting

At this point we have a collection of L labeled events $\{(t_j, l_j, m_j)\}_{j=1}^{L}$, where t_j is the timepoint, l_j is the integer label identifying the spike type, and m_j is the primary channel for this spike type. So far we haven't been trying to *explain* or *fit* the data array, but rather we were just detecting likely events and clustering them based on similarity. Ideally we would like

$$Y_{\texttt{white}}(m,t) \approx \sum_{j=1}^{L} W_{l_j}(m, t - t_j),$$

where W_k is the average spike type template as defined above. But we haven't been using this formulation in any way. In fact, the detections of two events $(t_{j_1}, l_{j_1}, m_{j_1})$ and $(t_{j_2}, l_{j_2}, m_{j_2})$ are completely independent when $m_{j_1} \neq m_{j_2}$ even when the timepoints are equal or very close.

In this fitting stage, we use a greedy fitting approach to reduce to a subset of the labeled events that best explain the whitened data. The implementation of the algorithm is a bit tricky, and involves several passes, and efficient storage of various computed quantities. For now I just state the inputs and outputs:

```
Algorithm 2: Fit
  Input: $Y_{\texttt{white}}$ - $M \times N$ array of raw data
  Input: S=$\{(t_j, l_j, m_j)\}_{j=1}^{L}$ - a collection of labeled events
  Input: $\{W_1, \ldots, W_K\}$ - spike templates
  Output: $\tilde{S} \subset S$ - a subset of the labeled events that best explain the
          data
```

4 Installation

The core MountainSort software is written in Qt5/C++. It requires Qt5 development environment to be installed on your system. For now MountainSort runs on Linux. It should work on Mac without too much trouble. Windows is a pain.

See the README.txt file for detailed instructions.