Andrew Robbins

MindMouse

Project Description:

Emotiv EPOC Neuroheadset

MindMouse is an application that interfaces the user's mind with the computer's mouse functionality. The hardware that is required for MindMouse is the Emotiv EPOC Neuroheadset (pictured on the right). This headset contains 14 EEG sensors that are used for measuring the user's brainwaves. The user must train MindMouse to recognize thoughts corresponding to mouse actions, such as left-click or scroll-up, to create a user profile. Once training samples are collected, a model that detects trained thoughts is created and MindMouse is ready for use.

The advantage of MindMouse over other computer pointing devices is that it is hands-free. The main benefit that I hope to achieve with MindMouse is for people with disabilities that prevent them from using a conventional mouse. Here the hands-free capability may enable them to use a pointing device where other options are not available.

This project is written in C++ and uses the following Libraries: LibSvm, kissFFT, BOOST File System, and Emotiv Research Edition SDK.

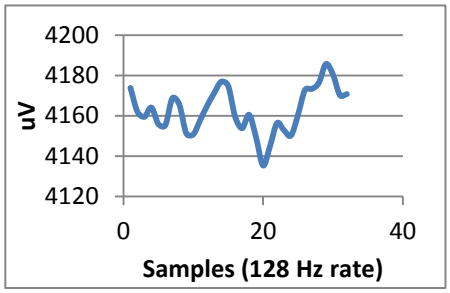
Collecting training data

MindMouse aims to distinguish the EEG readings from a particular user that correspond to trained actions, and to recognize when the user does not intend for MindMouse to take any action. To achieve this, MindMouse must first collect EEG data from the user that will become the training examples for the learning algorithm.

To input training examples to this system, the user goes to the training menu and selects an action to train. The user is required to train neutral, and some number of mouse actions. Training results in a few seconds of waves that are recorded and saved. The user can record as many examples as desired to get an arbitrarily large training set. During model creation, features from the trained action will become the positive training examples, and features from all other trained actions and from neutral will become the negative examples.

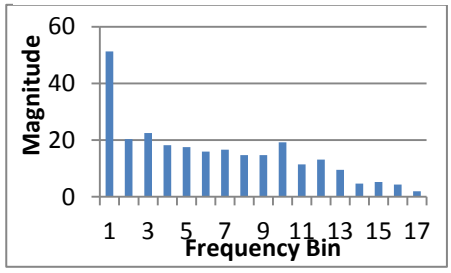
Feature creation

The input data to this system is 14 waves, measuring brain activity, each sampled at 128 Hz[1]. My feeling is that for the purposes of the machine learning algorithm, the values of the wave samples do not provide much information in such a raw form. This is partly because the wave is composition of a number of waves of varying frequencies. Additionally, I believe that the phase of the wave within the window that I am sampling it does not provide information useful for classifying thoughts. I want to know that sensor N is picking up a comparatively strong signal around 20 Hz, but I don't want to know where my sampling window happened to overlap this 20 Hz signal.



1/4 second of raw EEG samples from one sensor

To remedy this, I need to preprocess my input data. To do this I take 1/4 of a second worth of samples from each wave. Then I transform each wave into the frequency domain using a Fast Fourier Transform. I am using the kissFFT library for this transformation. After transforming the input data, I end up with 17 frequency bins, representing the magnitude of the wave within the corresponding bin. The bins contain the magnitudes of frequencies ranging from DC up to 64 Hz (i.e. the Nyquist limit given 128 Hz sampling rate). These frequency bins are the features for my learning algorithm (17 bins * 14 sensors = 238 features).



Features for the learning algorithm created from the wave above. Bin 1 is the DC component, bin 17 is 60-64 Hz. (The magnitudes are shown on a logarithmic scale.)

Creating a model to classify thoughts

To create a model of each action, I train a support vector machine. I am using LibSVM for the SVM implementation. I create a separate model to detect each trained action. There are two reasons for this, first I want to be able to select a distinct feature set for each trained action that best distinguishes the target action from all of the others. I believe that having separate feature sets for each action is better than trying to find one in common because it could be the case that one action has a strong correlation with a particular feature, but other actions show no correlation with that feature. The second reason for this is that I want to be able to detect multiple actions at once.

Feature selection is a critical piece of this application. During the course of this project, I found that I was faced with two competing objectives:

1. Enable the computer to automatically find correlations in brain waves that can identify a trained action.

[1] To interface with the headset I am using the research edition SDK which allows direct access to the raw EEG data.

2. Finish feature selection in a reasonable amount of time.

To achieve a balance between these objectives, I tried a number of different things before finally settling on one. Most of them did not work well enough for one reason or another. I will mention some of the things that I tried and explain what I finally settled on.

For my first attempt at finding a good feature set I implemented backward search. My thought was that the set of all of the features will contain the correlations that distinguish different actions. So I began my search with all of the features, trained the model with all but one of them, and ejected the feature where the 10 fold cross validation error was lowest when trained on the rest of the features. To do the cross validation I used the library's built-in cross validation function. There was a problem here: the built in cross validation function creates random bins each time it is called, and when I am searching over 200 features, I end up choosing features that happened to get a fortuitous cross validation bin allocation. So I was not converging to the correlated feature set that I wanted. To prove that this was what was happening, I ran cross validation 5 times for each feature set, and averaged the error. Afterward I saw that backward search was converging to low cross validation error. But… It took 8 to 12 hours to run it on a moderately sized training set.

I needed to keep the cross validation bins constant in the inner loop of my backward search, so that I could compare feature sets' cross validation error apples to apples. However, I don't want my whole search to be dependent on one chance bin setting. So I created an implementation of 10 fold cross validation that would distribute positive and negative training examples uniformly into bins, and in a deterministic order. Then in the outer loop of my search (after selecting each feature) I randomize the order of my training examples. This way I achieved apples to apples comparison between features at each step of the search, and still benefitted from randomized cross validation bins. This achieved a 5 times speedup compared to my previous backward search, but it was still slow. Happily, my processor runs 8 threads in parallel, so I split the inner loop of the search into 7 parallel threads. Together these two optimizations achieved a 35 times speedup.

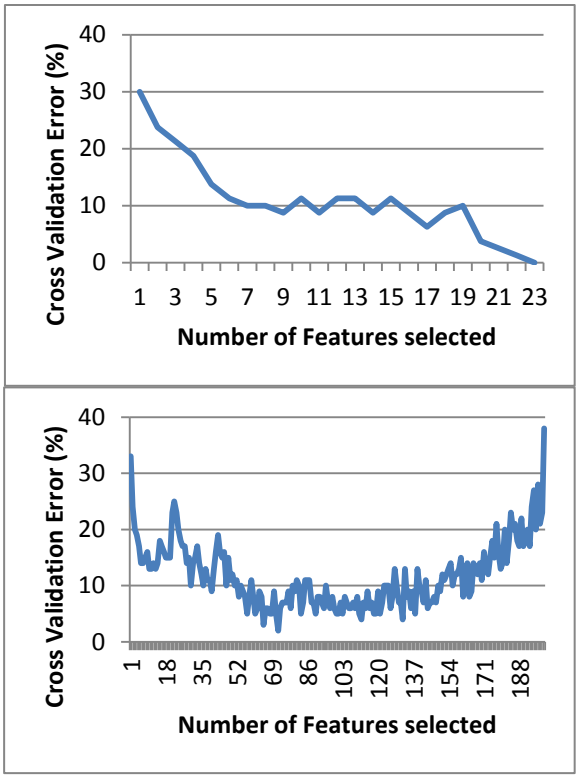
At this point I wanted to know if forward search would also converge to low cross validation error. So I implemented forward search similarly to my implementation of backward search, and I found that it also converged to low cross validation error with a similar number of features. But forward search reached low cross validation error much faster than backward search, in my runs, so I switched to forward search and I allow the user to stop the search early (and resume later if desired).

Now that I had something working, I wanted to try different kernels. Up to this point, I had been using a linear kernel, so I switched to using a Gaussian kernel. The Gaussian kernel had a parameter gamma, corresponding to $1/2\sigma^2$, and I tried a wide range of settings for this (from 0.001 to 65.536, doubling it each time). When using a Gaussian kernel, I noticed that forward

search would exclusively choose DC components from each of the sensors, until it chose all 14 of them. Afterward it would begin choosing components of non-zero frequency, and the cross validation error would quickly rise and not recover. So a Gaussian kernel did not work for this application. Next I tried a polynomial kernel of degree 2 (the constant had a value of 1.0). It took much longer to reach convergence each time I trained a model using the polynomial kernel; forward search would have taken days to complete. So for practical considerations, the polynomial kernel also would not work. Since the linear kernel worked and was fast, I felt it was the best practical choice, so I am using the linear kernel in this application.

Once I had created a model with low cross validation error over my training set, the next step was to see if this low cross validation error indicated that the model could predict when I was thinking the trained thought. When testing this, I found that if the time between when I inputted the training data to the time when I tested the response was not very long that the probabilities reported by the model were strongly correlated with the trained action. However, when I used the same model the next day, it was no longer very good at predicting the trained action[2].

My theory on this was that the SVM was being trained on my physiological state, such as my heart rate, at the time the training samples were taken. It was mentioned in class that ICA can be used to remove unwanted artifacts caused by eye movement and heartbeat from EEG data. So I looked into implementing artifact removal from my input signals. The algorithms that I came across that would automatically remove ocular artifacts required additional sensors near the eyes[3], so that after the raw data is converted into independent components, the components that correspond to eye movement can be detected and zeroed out. Since my hardware does not include such sensors, I couldn't implement that algorithm for this project. However, I felt that it was necessary to remove the contribution of eye movement and heartbeat from my input data, so I decided to constrain the feature search to use only



The top graph shows the cross validation error on a training set after each step of forward search, when features from all frequency bins were used. The bottom graph shows forward search over the same training set when frequencies below 4 Hz were excluded from the feature set.

[2] The model is identical each time it is used, because it is saved to disk and loaded when needed. Also, for testing purposes, I am using motor control thoughts like "clench fist" so that I can be confident that I am properly reproducing the thought.

[3] Joyce, C., Gordonitsky, I., Kutas, M. (2004) Automatic removal of eye movement and blink artifacts from EEG data using blind component separation. *Psychophysiology, 41,* 313-325.

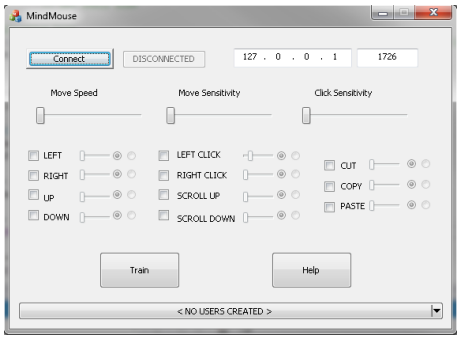
the features whose frequency was higher than 4Hz.[4] After making this modification, I found that the models that are created are more robust to changes in my energy level. I have two graphs above that show the performance of forward search before and after making this change. When the lowest frequency features are used, forward search quickly reaches 0% cross validation error, but seems to be learning about features that were not intended to be taught. When constrained to higher frequencies, forward search's minimum cross validation error was 2%, but the models that are produced seem to work much better.

Operation:

Once models are created and MindMouse is enabled, it collects EEG readings at a rate of 128 samples per sensor per second. MindMouse keeps a buffer of the most recent ¼ second of samples, and 32 times per second this buffer is converted into a feature vector and sent through each model. When MindMouse detects that 3 consecutive feature vectors are given a probability of indicating a trained action higher than the user configurable sensitivity, mind mouse sends the associated mouse command into the system.

Future Work:

I developed MindMouse as a Win32 console application for the purpose of CS229. I have begun construction of a Windows GUI that will encapsulate the functionality contained by the console application in a user friendly way. This GUI is shown on the right. The user interface will allow the user to enable/disable particular mouse functions, adjust sensitivities, and collect training data corresponding to mouse actions. Each action has a slider that will display the computer's belief about the probability that the user is thinking the particular action. This allows the user to adjust sensitivity and to retrain individual mouse behaviors as necessary. Also any subset of functionality can be enabled as desired by the user.



MindMouse user interface

[4] In the algorithm proposed by (Joyce, Gordonitsky & Kutas, 2004), one step that they take is to remove components with high power in the low frequency range that correlate to signals from sensors over the eyes. Also it seems likely to me that signals corresponding to heartbeat would be low frequency signals. Since I don't have sensors to correlate the unwanted signals, I believe that it is better to throw away the low frequency features than to have the model find correlations with unwanted artifacts that don't correspond to the trained action.