

Emotiv Experimenter

An experimentation and mind-reading
application for the Emotiv EPOC

Michael Adelson

Submitted for two-semester independent work (COS 497 and 498).

Faculty Advisor: Ken Norman

Second Reader: Robert Schapire

4/15/2011

1 Abstract

This report describes the development and features of Experimenter, an application based on the EEG capabilities of the Emotiv EPOC headset. As a research tool, Experimenter allows a variety of experiments based on classic stimulus-presentation paradigms to be run using the Emotiv. Unlike most EEG setups, however, Experimenter not only records data but also attempts online analysis and classification of the incoming data stream. With the proper experimental setup, then, Experimenter can be used as a simple mind-reading application. Experiment and application design, sample procedures, classification techniques, results, and technical details are discussed.

2 Honor Code

This thesis represents my own work in accordance with university regulations.

Michael Adelson

3 Table of Contents

1	Abstract
2	Honor Code
4	Introduction
4.1	EEG
4.2	The Emotiv Headset
4.3	Previous Work
5	The Experimenter Application
5.1	Initial Work
5.2	Online Classification Capabilities
5.3	Experimental Setup
5.3.1	The Basic Experiment
5.3.2	Justification
5.3.3	Variations
5.3.4	A Note about Timing
5.4	Features and Design

[5.4.1 Goals](#)[5.4.2 User Interface Design and Functionality](#)[5.4.3 Software Design](#)[5.4.3.1 Platform Choice](#)[5.4.3.2 Interfacing with Emotiv](#)[5.4.3.3 The Yield-View Presentation System](#)[5.4.3.4 The Parameter-Description System](#)[5.4.3.5 Timing Details](#)[5.4.3.6 Third-Party Software](#)[6 Experiments](#)[6.1 Eyes Open vs. Eyes Closed](#)[6.2 Faces vs. Places](#)[6.2.1 Version 1](#)[6.2.2 Version 2](#)[6.2.3 Version 3](#)[6.3 Artist vs. Function](#)[6.4 Faces vs. Expressions](#)[7 Data Analysis](#)[7.1 General Issues and Techniques](#)[7.1.1 Artifact Detection](#)[7.1.2 Time Series Classification](#)[7.1.3 EEG Classification](#)[7.1.4 Online Classification](#)[7.1.5 Voltage-based Preprocessing](#)[7.1.6 Time-Frequency Analysis](#)[7.1.7 Linear Discriminant Analysis](#)[7.2 Classification Algorithms](#)[7.2.1 K-Nearest Neighbors \(k-NN\)](#)[7.2.2 Penalized Logistic Regression \(PLR\)](#)[7.2.3 Decision Stump](#)[7.2.4 Voted Perceptron](#)[7.2.5 AdaBoost](#)[7.2.6 AdaBoost and Voted Perceptron](#)[8 Results](#)[8.1 Eyes Open vs. Eyes Closed](#)[8.2 Offline Validation](#)[8.3 Faces vs. Places](#)

8.3.1	Version 1
8.3.2	Version 2
8.3.3	Version 3
8.4	Artist vs. Function
8.5	Faces vs. Expressions
8.6	Overview
9	Discussion
9.1	Using the Emotiv Headset
9.2	Future Directions
9.2.1	Experiments
9.2.2	Experimenter Functionality
9.2.3	New Projects
9.2.3.1	Image Classifier
9.2.3.2	Thought-meter
9.3	Conclusion
10	Acknowledgements
11	References
12	Appendices
12.1	Appendix A: Additional Figures
12.2	Appendix B: Walkthrough
12.3	Appendix C: Experimenter Documentation

This report begins with a general overview of EEG (4.1) and previous work with the Emotiv headset (4.2 and 4.3). It then goes on to describe the Experimenter application itself. This includes a detailed description of the basic experiment format (5.3) and a summary of application functionality from the perspective of a user (5.4.2). The software design section (5.4.3) contains technical details regarding Experimenter's implementation; this section is intended primarily for those who might wish to extend Experimenter or to build their own applications using Emotiv. The experiments section (6) describes specific experiments run as part of the current project, and contains instructions for implementing these experiments using the Experimenter application. The data analysis section details considerations and techniques for working with Emotiv's EEG data (7.1) as well as various machine learning algorithms which were used for classification (7.2). This is followed by a summary of the results of applying this analysis to various collected data sets (8). Finally, I comment on my experience using the Emotiv headset

(9.1) as well as future directions for this work (9.2). The walkthrough in appendix B (12.2) provides a visual tutorial for aspiring users of Experimenter, while the documentation in appendix C (12.3) is aimed at providing further assistance to developers.

4 Introduction

The Emotiv EPOC headset is a cheap and easy-to-use mass-market EEG recording device. Targeted at video gamers, Emotiv promises to add a new dimension to the gaming experience by providing brain-computer interface (BCI) functionalities. Emotiv's simplicity has also attracted the interest of neuroscientists. Unlike traditional EEG setups and functional MRI, the Emotiv headset offers the possibility of running inexpensive experiments from the comfort of one's laptop. In the interest of exploiting these capabilities, I developed Experimenter, an application which can be used to collect and process EEG data within a configurable experimental framework.

4.1 EEG

Electroencephalography (EEG) is a nonintrusive means of recording brain activity. An array of electrodes attached to the scalp via conductive gel or paste is used to detect electrical activity produced by neurons firing within the brain. The resulting waveforms, popularly referred to as *brain waves*, can be used to diagnose epilepsy and some other abnormal brain conditions. EEG is also of interest to researchers because it can be used as part of an experiment to study the neurological activity following the presentation of specific stimuli or under specific conditions.

4.2 The Emotiv Headset

The Emotiv headset is an attempt to bring EEG capabilities to the masses. At \$299.00 (\$750.00 for the research SDK), the headset is relatively affordable. More importantly, the headset's fixed sensor-array makes it easy to use and highly portable. Since it connects wirelessly to any PC via a USB dongle, the user maintains a full range of motion. Instead of gel or paste, Emotiv simply requires that the user wet its foam-tipped sensors with contact lens solution prior to each use. The headset does pay a price, however, for such simplicity: there are only 14 channels, less than the number used in many research setups (Newman & Norman, 2010) (Parra, Spence, Gerson, & Sajda, 2005). The data sample rate, at 128Hz, is also far lower

than in more powerful setups which sample at up to 1000Hz (Newman & Norman, 2010) (Parra, Spence, Gerson, & Sajda, 2005). The arrangement of the electrodes approximates the popular 10-20 positioning scheme (Emotiv, 2010).

Although the headset is an EEG device, most developers using the device do not have access to the raw EEG data stream (this is available only in the research edition) (Emotiv, 2009). Instead, the Emotiv APIs exposed by the standard SDK consist of three event-based classification suites. The ExpressivTM suite tries to detect the wearer's facial expression, the AffectivTM suite tries to detect the wearer's mood, and the CognitivTM suite tries to detect occurrences of user-defined cognitive events (Emotiv, 2010).

Emotiv's online application store offers a number of innovative EEG-powered programs built on these suites. EmoLens tags users' photos based on their emotions when viewing them. NeuroKey acts as a virtual keyboard for the disabled which can be interfaced with via both facial expressions and trained cognitive actions. The Spirit Mountain game promises a true fantasy experience by allowing the user to manipulate a virtual world with his or her mind. The headset has also become a part of several interesting side projects, such as acting as a controller for a superman-style human flight simulator and a robotic car (Solon, 2011) (Squatriglia, 2011).

Customer reviews of the device, however, are highly mixed. The facial expression-detection system is said to work very well, but unsurprisingly what reviewers really care about is the headset's promise of mind-reading capabilities. In this area, most reviewers seem to feel that the Emotiv's thought detection algorithms are at best unreliable, making it nearly impossible to play even simple games like Pong (Root, 2011) (Dakan, 2010). My own experience with Emotiv's free Spirit Mountain demo game seemed to confirm these reviewers' experiences. In the game, the player's first task is to drive away evil spirits by clenching his or her teeth and growling. This works perfectly. The next set of tasks require the user to train Emotiv to recognize two distinct thought patterns: *lift* and *pull*. After training, I could never seem to trigger lifts or pulls with conscious thoughts. More frustratingly, Emotiv often detected lifts and pulls when my attention was focused elsewhere! However, some sources suggest that more practice may help eliminate such difficulties (Emotiv, 2010).

4.3 Previous Work

Lillian Zhou (COS 11) worked with the Emotiv headset for a one-semester independent work project last spring. Lillian ran several experiments with Emotiv, one manually (running the Emotiv in the background to collect data while instructing the subject to perform a task) and one with a simple integrated presentation and recording loop using Emotiv's native C++ API (Zhou, 2010). The difficulty of building and running experiments in this manner demonstrated the necessity of a full-featured application devoted to supporting Emotiv's use as a research tool, particularly for the purpose of allowing non-programmers to experiment with the device.

5 The Experimenter Application

This section describes the design and feature set of Experimenter. At the time of this writing, the full application source code, source-level documentation, and executable binary are available for download at <http://www.princeton.edu/~madelson/experimenter>.

5.1 Initial Work

The first iteration of the Experimenter application focused on supporting on supporting three basic features:

1. Allowing folders of images to be loaded into the application and treated as classes of stimuli.
2. Allowing the user to run an experiment where images from two such classes are displayed to the user.
3. Collecting, tagging, and logging the raw EEG data coming from the Emotiv headset while the images were being displayed,

These capabilities enabled simple stimulus presentation experiments (6.2.1), generating data for offline analysis (8.3.1). Implementing this basic application also provided insight into design challenges such as properly interfacing with the Emotiv headset and supporting a high degree of configurability. These issues were given special consideration when redesigning the code base for the final version of Experimenter.

5.2 Online Classification Capabilities

While the ability to collect data via such experiments was useful, the initial experimenter application lacked the ability to process the incoming data online as it was being collected. The next iteration of the application thus focused on combining the application's presentation and data-collection capabilities with a classification tool chain that could be run in an online manner. Online classification of EEG signals is of particular interest when working with the Emotiv, since the device is sufficiently user-friendly and portable to be used in practical BCI scenarios. Attempting online mind reading had the added benefit of making experiments more fun for subjects. This help to maintain attentiveness, which in turn may improve the quality of the collected data.

While any successful online classification could be considered mind reading, a truly interesting mind reading scenario should use this capability in a way that convinces the user that mind reading is really taking place. In other words, such a setup must enable the user, without looking at the application code, to believe within reason that the application really is relying on EEG data to make its predictions (as opposed to other sources of information to which it is privy). Such a setup can be said to achieve *subject-verifiable mind reading*. To more clearly illustrate this concept, imagine an application that displays either a face or a scene image to the user and then guesses at the type of image displayed based on online classification of collected EEG data. Assuming that the classifier obtains above-chance performance, this application possesses some basic level of mind reading/BCI capabilities. However, few skeptical users would be convinced of this, since it would be easy for the application to cheat because it knows the type of displayed image and thus could succeed at classification without ever examining the EEG output. There are other less-dishonest ways in which the mind reading illusion could be broken by poor experimental design. As discussed in section 7.1.1, the Emotiv headset is very sensitive facial muscle movements. If an experiment were to trigger such movements in a systematic way, it runs the risk of introducing a muscle-based signal that could allow the application to classify the incoming EEG data without actually using neuronal information. Thus, the application's capabilities may be written off as something more akin to facial expression recognition rather than true mind reading. Furthermore, the data collected using such a design would be of limited offline use due to contamination by these so-called motion artifacts. Incidentally, this can be seen as a problem with Emotiv's CognitivTM classification suite. Since users are given little guidance as to what to do when training the classifier to recognize a particular cognitive event, it

is difficult to tell whether the resultant classifier really relies on the neuronal signal rather than using stronger muscle-based signals.

To achieve subject-verifiable mind reading then, it is necessary to present the user with a decision whose outcome cannot be known to the application except possibly through collected EEG data. However, this choice should also be structured in a way that does not invite subjects to accidentally contaminate their data with systematic muscle-based electrical activity. The experiment described in the following sections was designed to adhere to these guidelines.

5.3 Experimental Setup

Experimenter allows users to execute a highly configurable stimulus presentation experiment with support for real-time classification in addition to data collection. Furthermore, each stimulus display is followed by a simple interactive task to improve subject concentration.

5.3.1 The Basic Experiment

The following table summarizes the key experimental parameters:

Parameter	Description	Typical Value
S_1 and S_2	Two classes of stimuli (images or text) used during the experiment. The stimuli in each class are classified as belonging to one of two specified subcategories.	S_1 consists of 100 face images, where 50 are male and 50 are female. S_2 consists of 100 scene images, where 50 are indoors and 50 are outdoors
T	The stimulus display time (in ms)	500
N_{train}	The number of training trials for each stimulus class	50
N_{test}	The number of test trials	100
C_1, C_2, \dots, C_n	A set of classifiers to be used in online classification	C_1 is an AdaBoost classifier (see 7.2.5) with decision stump (see 7.2.3) as the weak learner

The experiment proceeds as follows:

1. Assemble two lists of images, L_1 and L_2 . L_1 consists of the repeated concatenation of the set of stimuli in S_1 , with each concatenated set randomly shuffled. L_2 is constructed in the same way using the stimuli in S_2 . Each experimental trial will consist of the presentation of one stimulus from each list, so both lists are of sufficient length so as to contain one stimulus for each training trial and

each test trial in the experiment. The practice of exhausting all available stimuli in a class before allowing for any repetition rather than just randomly sampling from the stimulus set with replacement is desirable because a subject's reaction to a stimulus may be different on the first viewing than on subsequent viewings. However, some experiments may have fewer stimuli than trials and thus require repeat presentations. In this case, shuffling each concatenated set of stimuli helps eliminate any potential bias due to the subject's becoming aware of patterns in the presentation (e. g. that the same two stimuli are always presented together) while still maintaining a (probabilistically) long time period between repeated presentations of a stimulus.

2. Training Phase (repeat $2N_{\text{train}}$ times)

- a. Let I_1 and I_2 be the next stimuli in L_1 and L_2 , respectively. Let I^* be randomly selected to be either I_1 or I_2 , unless doing so would cause either I_1 or I_2 to be selected more than N_{Train} times (in this case the stimulus from the under-selected class is chosen to be I^*).
- b. Randomly assign one of the images to be displayed on the left side of the screen. The other will be displayed on the right.
- c. On each side of the screen, display the class name of the image assigned to that side of the screen. In the center of the screen, an arrow indicates which side of the screen will contain I^* . At this point, the subject should focus on the indicated side of the screen (Figure 7).
- d. Replace the class names with fixation crosses, and display for a brief period of time (Figure 8).
- e. Display the images I_1 and I_2 are on their respective sides of the screen for time T (Figure 9).
- f. Present the subject with a display consisting of a question corresponding to the class of I^* (e.g. *Was the image of a male face or a female face?* if I^* was a face image). The subject may answer with either of the class's two subcategories (e.g. *male* or *female*), or with a third option: *I don't know (invalidate trial)*. This option allows subjects to throw away trials where they accidentally looked at the wrong stimulus (Figure 10).
- g. If the subject answered *I don't know (invalidate trial)*, or if the subject chose a subcategory that did not match I^* 's subcategory, or if a motion artifact was detected during the trial (see 7.1.2), repeat the current trial with new stimuli (go to 2a).

- h. Record the trial (go to step 2).
3. C_1 through C_n are trained on the trial data collected during the training phase.
4. Test Phase (repeat N_{test} times)
 - a. Let I_1 and I_2 be the next images in L_1 and L_2 , respectively.
 - b. Randomly assign one of the images to be displayed on the left side of the screen. The other will be displayed on the right.
 - c. On each side of the screen, display the class name of the image assigned to that side of the screen. In the center of the screen, a bidirectional arrow indicates that the subject may focus on either side of the screen. At this point, the subject should choose one side of the screen (and thus one stimulus class) to focus on (Figure 11).
 - d. Replace the class names with fixation crosses, and display for a brief period of time.
 - e. Display the images I_1 and I_2 on their respective sides of the screen for time T .
 - f. Use each classifier to predict which class of image the subject looked at. Display these predictions (Figure 12).
 - g. Present the subject with a display asking which class of image she focused on ().
 - h. Based on the subject's response to g, present the subject with a display consisting of a question corresponding to the class of the selected image. Again, the subject may answer with either of the class's two subcategories, or with *I don't know* (invalidate trial) (Figure 12).
 - i. If the subject answered *I don't know* (invalidate trial), or if the subject chose a subcategory that did not match the selected image's subcategory, or if a motion artifact was detected during the trial, repeat the current trial with new images (go to step 4a).
 - j. Record the trial (go to step 4).

In addition to the parameters specified above, many other variables can be configured through Experimenter's interface, including the size of the displayed images, the amount of time for which the fixation crosses and instructions are presented, and the rest period between trials. Online classifiers can also be configured to periodically retrain during the test phase in an effort to improve performance.

5.3.2 Justification

This experimental design can be seen as an instance of subject-verifiable mind-reading because the user makes a mental choice during the test phase, the result of which is not known by the application until after it has made its prediction. Although the need for the subject to view different sides of the screen might seem to be a source of motion-artifacts, any movement of this sort should only occur between trials and thus should not corrupt the data. Even if such corruption were to occur, the randomized placement of the images prevents this from being a source of systematic bias.

5.3.3 Variations

Experimenter also contains options to run several variations on the basic experiment. For example, rather than placing the images at opposite sides of the screen, the images can be superimposed with the top image made semi-transparent. The subject's job, then, is to focus on one particular image. This feature also allows for experiments in which the task varies between conditions but the stimulus does not. To implement such a design, the same set of stimuli can be used for both classes with the experiment run in superimposed mode at zero transparency (as in 6.3). Another useful option is the ability to have Experimenter play a tone at the end of each trial. This allows experiments to be run where the subject does something other than look at the screen (as in 6.1). Experimenter also supports the use of text in place of images, which permits experiments regarding the subject's response to particular words as well as text-based tasks. Finally, the policy of invalidating trials where the subject responded incorrectly to a question can be disabled, as can the asking of the question. This is useful when there is no definite answer to a particular question (e. g. "Would it be hard to illustrate the given text?").

5.3.4 A Note about Timing

The trial timing supported by Experimenter is not millisecond-accurate. This means that a stimulus which is set to display for 500ms may display for a slightly longer or slightly shorter period of time (see 5.4.3.5). The data recorded during this time, however, maintains the time stamp generated by the Emotiv. Thus, the accuracy of the data time stamp is not subject to the inaccuracy of the display timing. To guarantee that a minimum amount of time's worth of data is recorded, it is generally sufficient to simply pad the display time value by 50ms.

5.4 Features and Design

This section gives an overview of the Experimenter application's capabilities and feature set. See appendix B (12.2) for a visual tutorial for the application.

5.4.1 Goals

Experimenter was designed with the purpose of being useful even to those with no desire or capability to modify its code base. This required making the program highly configurable yet (hopefully) easy to use. Furthermore, no attempt was made to replicate the capabilities offered by the Emotiv TestBench application which is provided with the research SDK and thus is presumably available to any user of Experimenter. Those features include real-time plotting of EEG data as well as a connectivity display, both of which are vital for getting the headset properly situated.

5.4.2 User Interface Design and Functionality

The Experimenter graphical user interface (GUI) is partitioned into three distinct regions, each of which allows for the configuration of one part of the experiment. The *Experiment Settings* region allows users to configure general experiment parameters, such as the image display time, as well as output options such as raw data output or an experiment log (Figure 13). The *Classifiers* region allows users to create and configure machine learning algorithms for online classification (Figure 14, Figure 15). Both classifier-specific parameters and general feature selection parameters can be adjusted. Multiple differently-configured classifiers can be run in parallel during an experiment to compare their performance. The region also contains a panel for configuring the artifact detection algorithm (Figure 14). When the headset is connected, artifact detection is performed in real time, giving the user feedback as he or she tunes the various parameters. Finally, the *Classes* region allows the user to create and configure classes of stimuli (Figure 16). Creating a new stimulus class from a folder of images and/or text files is simple. The class is created either by selecting its folder via the *New* button or by dragging and dropping the folder into the list of classes on the *Classes* tab (multiple folders can be loaded simultaneously in this manner). This action creates a new tab for the class, where various settings such as the class's name and EEG marker (used to tag data when a stimulus from that class is being displayed) can be configured (Figure 17). The class's tab also allows the folder's stimuli to be viewed and either included or excluded from the class. The *Classify*

button launches a tool for quickly specifying the subclass (e. g. male or female for face images) of each stimulus (Figure 18).

◆◆◆◆◆◆◆◆◆◆ To make configuration easier over multiple sessions, each configurable region supports *save* and *load* functionality. This is especially valuable for large stimulus classes. Each stimulus can be tagged with a subclass once, and the result can then be saved to a special file in the folder where it will be accessed automatically when the class is loaded in the future.

5.4.3 Software Design

This section details several major design choices and patterns in the Experimenter code base. It is intended both to justify the software design and to act as a high-level guide for those interested in using or modifying the code.

5.4.3.1 Platform Choice



Experimenter is written completely in the C# programming language which runs on Microsoft's .NET framework. The primary reason for choosing C# for this project is that Emotiv officially supports a C# wrapper for its native C++ SDK. C#'s automatic memory management and built-in support for graphical components make it well-suited for building graphical desktop applications like Experimenter. However, its use is restricted to .NET which in turn is generally restricted to the Windows operating system. Currently, this is not an issue since Emotiv only supports Windows (Emotiv, 2010). Should Mac or Linux support become available, though, porting the application is not completely out of the question because the open-source Mono implementation of .NET supports these platforms (Novell, 2011). Another disadvantage of .NET is a seeming lack of freely available numerical libraries for the .NET framework. On the other hand, C#'s lambda functions, extension methods and language integrated query (LINQ) features are extremely convenient for the sort of data-shuffling used in preprocessing and formatting data for classifier input (Figure 19).

5.4.3.2 Interfacing with Emotiv


The research edition of the Emotiv SDK exposes an API for accessing the headset's raw data feed. This interface exposes a strange mixture of event-driven and procedural access methods and is generally difficult to work with. Once the headset becomes connected, incoming raw data is placed in a

fixed-size buffer until it is explicitly polled by the application. If this is done too infrequently, perhaps because lengthy processing (e. g. classification) occurs on the polling thread between polls, some data will be lost.

In an attempt to make the Emotiv easier to work with programmatically, Experimenter wraps the Emotiv API with an instance of the publish/subscribe design pattern (Freeman, Freeman, Bates, & Sierra, 2004). Consumers of EEG data are implemented as passive listeners who perform some action upon the arrival of data. These listeners subscribe to a data source object that periodically polls the data buffer and publishes the collected data to each of its listeners. These messages are delivered on threads that are owned by the listeners, thus keeping the polling thread free to check for new data. In Experimenter, the data source also alerts its listeners when it gains or loses its connection to the headset (Figure 20).

There are several advantages to this design. Because the polling thread of execution does not have to wait while each listener processes the incoming data, the risk of data loss is minimized. The ability to broadcast EEG data from one source to many listeners is valuable because Experimenter has several distinct consumers of this data. In the Experimenter GUI, separate listeners are used to implement real-time artifact detection and headset connection status; each listener is associated with a particular task and graphical component. While experiments are being run, several more listeners may be active to watch for a loss of connection, to log raw data, or to break incoming data into tagged trials for classification. Another benefit is the ability to substitute a mock data source that publishes a random signal in place of the Emotiv headset data source when the headset is not connected. This is vital for testing the application without constant use of the headset.

5.4.3.3 *The Yield-View Presentation System*

As the experiment grew more complex, so too did the requirements for its implementation. Since the experiments logic requires updating the display based both on time and user input, the simple timer-based implementation that was sufficient for the initial implementation was no longer sufficient. While the experiment logic remained fairly simple to describe sequentially with standard control flow constructs (e. g. if-then-else and loops), running such code directly within the GUI thread of execution would have prevented that thread from being able to respond to user input, such as a button press or an attempt to close the application window. The simplest alternative was to map the logic onto a sequence of GUI events, with

timers and button presses triggering changes to the display as well as the creation of new timers and buttons which in turn would be used to trigger future changes. Such an implementation would have made the code extremely complex and unmaintainable, since it would lose all semblance of the sequential logic of the experiment.

Another issue was the proper management of various resources. Even though the .NET framework provides automatic memory management in the form of garbage collection, in some cases explicit management is still required. This is because some managed objects hold access to resources such as open files, images, and graphical components which are not *expensive* in the sense of consuming substantial process memory but which do consume substantial underlying system resources. The supply of such resources to a process is typically limited by the operating system. Since there is no guarantee that the managed handlers of these resources will be garbage collected in a timely manner (especially since the memory footprint of the managed object may be very small), it is considered good practice to explicitly dispose of these handler objects (Microsoft, 2011). This is of practical importance for Experimenter because the application uses a large number of these resources (typically for short periods of time) over the course of several experimental runs. A related issue is ensuring the proper disposal of the aforementioned subscribers to the headset data source. For example, it would be undesirable if a data logger from one experiment failed to unsubscribe properly before the next experiment, since this would corrupt the first experiment's data. In general, it is vital that the application be able to perform proper cleanup even when experiments terminate abnormally (e. g. the user stopped the experiment early). Experience suggests that this happens quite frequently, either due to time constraints or due to the discovery of a misconfigured setting. Thus, it would be unacceptable to have to restart the application or to have data corruption occur in such cases.

Experimenter's solution to these problems is a presentation system which makes heavy use of two powerful C# features. The *yield* keyword allows a function to return a sequence of values in a *lazy* manner. That is, rather than assembling the entire sequence and returning it to the caller, the sequence-generating code is suspended until the caller actually attempts to access an element in the sequence. At this point, the sequence-generating code executes until the next value is *yielded* to the caller, at which point it is once again suspended (in reality, the code executes normally but is transformed into a state machine by the compiler so as to provide suspension semantics). The *using* keyword provides a convenient

way to guarantee the proper disposal of objects which require explicit cleanup. Resources which need to be disposed can be declared within a *using* statement and then used within the corresponding *using* block. When the program exits the scope of the *using* block in any way, the object will be disposed.

The other cornerstone for the presentation system is the notion of a view. A view is like a slide in a presentation: it is deployed to the screen at which point it may display text, images, or other graphical components. At some point an event causes the view to finish, at which time it and all its resource-intensive components are properly disposed of. This logic is implemented in an abstract class which can be extended to implement specific behavior, such as displaying images, prompting the user for input, or drawing a fixation cross.

Within this framework, an experiment can be implemented by a function which *yields* a sequence of views. This allows the logic of the experiment to be expressed sequentially, with resource-intensive elements scoped by *using* blocks. In reality, the code is executed within an animation loop which deploys each view to the GUI thread of execution where it is displayed (Figure 21). All of the complexity of multithreading is handled at the animation loop and abstract view level, keeping it absent from the main experiment code. In addition to supporting the experiment functionality, the presentation system is also used to implement Experimenter's built-in tool for assigning subclasses to stimuli.

5.4.3.4 The Parameter-Description System

Allowing for a great degree of configurability via a graphical interface is challenging because each configurable parameter requires adding code for the appropriate type of input component (e. g. a text box) as well as a descriptive label and tooltip. Furthermore, reasonable default values must be provided and user input must be parsed and validated to ensure that it is within the allowed range. This sort of coding accounted for a large percentage of the work in developing the initial version of the application, so a different solution was required to make implementing the current version feasible.

Experimenter's solution to this problem uses custom C# *attributes*, which allow code to be decorated with runtime-accessible meta-data. Rather than writing code to express the GUI logic for each configurable parameter, parameters are simply expressed as properties which are then decorated with *Parameter* or *Description* attributes. These attributes can be used to specify display names, descriptions, default values, and valid value ranges for the parameters. A custom component called a

ConfigurationPanel can then use this information to dynamically construct a graphical component that allows the user to configure the parameters. Not only does this system save considerable programming time, but also it organizes the parameter specifications separately from the GUI logic, making the code neater and easier to modify (Figure 22). Usage of these attributes to annotate classes throughout the code base also allows many common functions to be implemented generically (i.e. in a way that allows them to operate on many types of objects). For example, *Experimenter* contains and uses generic functions for copying, instantiating, error-checking, and logging objects based on their parameters.

5.4.3.5 Timing Details

As previously mentioned, *Experimenter* does not support millisecond-accurate timing of presentation displays. This is due to the use of the *System.Windows.Forms.Timer* class, whose accuracy is limited to 55ms (Microsoft, 2011). The reason for *Experimenter*'s use of the *System.Windows.Forms.Timer* instead of, for example, the more accurate *System.Threading.Timer*, is that the former ticks on the GUI thread of execution (which is convenient when updating the display), while the latter uses thread pool threads. Thus to update the display upon the ticking of a *System.Threading.Timer* requires invoking a function on the GUI thread, which carries its own overhead, especially if the GUI thread is currently held up processing another event (indeed this may be a source of the *System.Windows.Forms.Timer*'s inaccuracy).

While this level of inaccuracy may seem problematic, in most cases the accurate time stamping and tagging of the data is far more important than is the accurate timing of the display, since data collected at the end of a trial can always be ignored. To ensure accurate tagging relative to what is actually being displayed, *Experimenter*'s *MCAEmotiv.GUI.ImageView* class sets the Emotiv marker (used to tag the next data point collected) immediately after the display control has been deployed to the window, and sets the marker back to the default value immediately before removing this control. To maintain accurate time stamps for collected data, *Experimenter* simply uses the original time stamps generated by Emotiv.

5.4.3.6 Third-Party Software

Although for the most part *Experimenter* is based on original code, the .NET Base Class Library, and the Emotiv SDK, it also makes use of open-source third-party software where possible. The Math.NET

Iridium library provides APIs for many common numerical procedures, such as the fast Fourier transform (FFT) and basic matrix operations (Math.NET Numerics, 2011). On the GUI end, the JacksonSoft custom tab control is used to implement the interfaces for configuring classifiers and stimulus classes (Jackson, 2010).

6 Experiments





Since the focus of this project was not to investigate the EEG response to a particular stimulus but instead to investigate the usage of the Emotiv headset for a general class of EEG experiments, I tried several different experiments over the course of the project. These experiments test both Emotiv and the capabilities of the Experimenter application.

6.1 Eyes Open vs. Eyes Closed

Determining whether a subject's eyes are open or closed is one of the few EEG classification tasks that can easily be performed by examining the voltage time series with the naked eye. When a person's eyes are closed, the EEG signals at the occipital electrodes (channels O1 and O2 on Emotiv) exhibit distinctive high-amplitude waves (Figure 23) (Zhou, 2010). Thus, a simple eyes open/eyes closed experiment provided a reasonable starting point for testing Experimenter's capabilities.

The experiment used the following format. During each training trial, the subject was presented with an instruction, either "Open your eyes" or "Close your eyes." The subject then followed this instruction until a tone was played, since a visual queue does not work in the closed condition. The subject then responded to a question asking whether he or she had just opened or closed his or her eyes. If the response was not consistent with the instructions, the trial was invalidated. During test trials, the subject was given a choice as to whether to open or close his or her eyes during each trial. The following setup steps implement this design using Experimenter (only changes from default options are listed):





1. Create two stimulus folders, each containing one arbitrary stimulus. In this case, images of an open eye and a closed eye were used. Load the folders into Experimenter to create classes.
2. Label the open eye class as "Open your eyes" and the closed eye class as "Close your eyes."
3. Set the question for both classes to be "Were your eyes open or closed?" and the answers for both classes to be "My eyes were open" and "My eyes were closed."

4. Classify the single stimulus in the closed class as My eyes were closed and the single stimulus in the open class as My eyes were open.
5. On the image display settings panel, select *Beep After Display*. Optionally set *Width* and *Height* to one to avoid actually displaying the placeholder stimuli.
6. In the experimenter settings panel, configure the experiment to use a relatively long *Fixation Time* and *Display Time* (e. g. 750ms and 1000ms respectively). This allows sufficient time to pass after the subject closes his or her eyes so that a steady state can be reached.
7. On the artifact detection panel, disable artifact detection.





6.2 Faces vs. Places

Another classic experimental paradigm is to try and distinguish between the neural processing of faces and scenes. The difference is known to be detectable, although traditionally such experiments are done with fMRI rather than EEG (Zhou, 2010).

6.2.1 Version 1

The initial version of the experiment supported only data collection as the subject viewed a series of face and scene images. The images were scaled to a size of 400 by 400 pixels and displayed in color for one second each, preceded by a randomly-timed fixation cross display and followed by a two-second break. There were several notable problems with this experiment. First, the experiment proved very tedious because of the lack of any requirement for user input. This made it very easy for the subjects mind to wander. Second, the long display time gave the subject time to move his or her eyes around the image, which could have introduced motion-related noise. Worse, if the artifacts of the motion were systematically different for the two classes of images, this could have provided the classifier with an invalid means of classification. Finally, displaying the images in color may have added noise or even systematic bias to the data as a result of the subjects response to particular colors.

6.2.2 Version 2

The first mind-reading version of the faces vs. places experiment attempted to resolve these issues using various features of the Experimenter application. In this version, the subject was presented with superimposed gray scale images from two Experimenter stimulus classes: Faces and Places.

During the training phase, subjects were directed to focus on either the face image or the place image before viewing the superimposed pair. During the test phase, the subject was allowed to choose mentally which image to focus on prior to the seeing the display. After viewing an image, the subject had to determine whether the image depicted a man or a woman (for faces) or whether it showed an indoor scene or an outdoor scene (for places). Images were displayed for 500ms (shorter display times of 200-250ms were also attempted). Typically, 50 training trials per stimulus class were used along with 100 test trials. Both myself and a test subject found this experiment to be frustrating because it was often difficult to avoid viewing both images (particularly since faces are very salient). Perhaps because of this, the classifiers I tried performed poorly under these conditions. The following setup steps implement this design using Experimenter (only changes from default options are listed):

1. Create two stimulus folders, one containing faces and one containing places. Load the folders into Experimenter to create classes.
2. Label the classes appropriately, and use the classification tool to assign male/female subclasses to the face images and indoor/outdoor subclasses to the place images.
3. On the image display settings panel, select *Superimposed* and tune the *Alpha* (transparency) parameter until both superimposed images are suitably visible.





6.2.3 Version 3

The final iteration of the faces vs. places experiment motivated the development of the basic side-by-side mode in Experimenter. In this version, the face and place images in each trial are displayed on opposite sides of the screen. During the training phase, the subject is directed to look at one side of the screen, and during the test phase he or she is allowed to choose a side to look at. Following setup steps one and two for the previous experiment implements this design using Experimenter.

6.3 Artist vs. Function

This experiment tested Emotiv and Experimenter with another classic paradigm: artist vs. function (Johnson, Kounios, & Nolle, 1997) (McDuff, Frankel, & Norman, 2009). In this experiment, the subject is shown a concrete noun for two to three seconds, during which he or she either (1) imagines drawing the object (the artist task) or (2) attempts to mentally list as many uses as possible for the object

(the function task). Before each training trial, the subject is instructed as to which task to perform, and before each test trial the subject is given time to mentally choose a task. After each trial, the subject must answer whether the noun would be easy or difficult to draw (for the artist task) or whether the noun has more than one function (for the function task). These answers are logged, but are otherwise disregarded. The following setup steps implement this design using Experimenter (only changes from default options are listed):

1. Generate a long list of concrete nouns, perhaps using the MRC Psycholinguistic Database (Wilson, 1988). Copy the word list into a text file with one word on each line.
2. Create two stimulus folders, each containing a copy of this text file. Load the folders into experimenter to create the classes.
3. Label the classes Artist and Function, and specify their questions and answers appropriately. Leave all stimuli as *unclassified*.
4. On the image display settings panel, select *Superimposed* and set *Alpha* to zero or 255. This guarantees that only stimuli from one class will show (this is fine since the classes use the same stimuli).
5. On the experiment settings panel, set the *Display Time* to 3000ms and the *Question Mode* to Ask.

6.4 Faces vs. Expressions

Like the artist vs. function experiment, this experiment attempts to distinguish between two very different types of processing. Subjects are presented with two superimposed stimuli: an image of a face and a simple mathematical expression. When directed to focus on the face, subjects must determine whether or not the person is male or female. When directed to focus on the expression, the subject must attempt to evaluate the expression in the given time. A display time of two or more seconds was used to give the subject sufficient time to at least make progress evaluating the expression. Since Experimenter currently supports only two possible subclasses per stimulus class, all expressions needed to evaluate to one of two values. In an attempt to keep processing as uniform as possible, I attempted to avoid answer values that might allow shortcuts to some expressions. For this reason, I chose 14 and 16, since these two values have the same parity and are close in magnitude. The following setup steps implement this design using

Experimenter (only changes from default options are listed):

1. Create a text file with one mathematical expression on each line. This can easily be done with code such as (Figure 25).
2. Place this text file in a stimulus class folder and load the folder into the application. Configure the class's *Answer* values to match the possible expression values.
3. Generate and load a face stimulus class as previously described.
4. On the image display settings panel, select *Superimposed* and configure the *Alpha* value for the image so that the text is clearly visible but does not dominate the image (I found that a value of 200 was appropriate).
5. On the experiment settings panel, set the *Display Time* to 2000ms.

7 Data Analysis

7.1 General Issues and Techniques

7.1.1 Artifact Detection

As emphasized by the success of Emotiv's AffectivTM suite, EEG signals can be used to detect the movement of facial muscles. Indeed, the detection of the so-called artifacts that result from these motions is critical in processing EEG signals because the powerful motion-related noise swamps the weaker neuronal signal (Figure 24). The most notorious such artifact is the eye blink, which causes a large deflection of the signal that is typically 20 dB above background activity (Parra, Spence, Gerson, & Sajda, 2005). Eye blinks are particularly problematic because they are often involuntary and are likely to result from staring at a computer screen. To maintain the integrity of its data, Experimenter implements a simple artifact-detection algorithm which is used to detect and exclude contaminated trials.

The algorithm is based on one used in an EEG study by Ehren Newman (Newman E. , 2008). It takes four numeric parameters Alpha_1 , Alpha_2 , Threshold_1 , and Threshold_2 and one EEG channel. First an exponential moving average of the voltage time series from the specified channel is computed using Alpha_1 . If at any time this average differs from the series mean by more than Threshold_1 , the trial is rejected. Otherwise, this moving average is subtracted from the time series and the algorithm computes a

second moving average using Alpha_2 on the residual signal. If the absolute value of this second average ever passes Threshold_2 , the trial is also rejected (Newman E. , 2008).

Since the alpha and threshold values used by Newman (0.025, 40, 0.5, and 40) generally work quite well with the Emotiv, these are the default values used by Experimenter (Newman E. , 2008). Experimenter uses AF3 as the default channel because that electrode's location on the subject's forehead causes it to react strongly to eye blinks.

Observation of the online FFT output provided by the Emotiv TestBench application suggests that motion artifacts could also be detected by watching for power surges at low frequencies in the signal's Fourier transform. Since the algorithm described above seems to be sufficiently effective, however, I did not explore this possibility further.

7.1.2 Time Series Classification

Data collected by the Emotiv headset, or indeed by any EEG setup, is in the form of a collection of a real-valued voltage time series (one for each scalp electrode). When considering how to analyze and classify such data, there are two distinct possible approaches. The first is a *steady-state* approach. With this approach, one attempts to develop a classifier which can categorize an arbitrary subsection of the time series. Since obviously the same signal may look quite different depending on where the sampling *window* is placed, some sort of preprocessing is required that can extract a set of features that are mostly independent of window placement. Such features can be said to represent steady-state properties of the EEG signal, and can thus be compared across samples. Possible options for such features include the mean voltage of the signal over the window, the average difference of the signal voltage from that mean, and the power spectrum of the signal.

The alternative approach is to keep time as a factor in the equation. In many cases, one wishes to classify the EEG response to some stimulus. Thus rather than building a classifier that operates on an arbitrary subsection of the time series, one can instead design a classifier to work with samples of data which have been aligned based on the time of the stimulus presentation. For example, if one collects many 500ms samples of data, each of which is time-locked to the presentation of some stimulus, then the voltage data at 200ms *post stimulus onset* can be treated as a feature that can be compared across samples. The argument for using this approach instead of the steady-state approach is that, in many cases, distinct

responses to certain stimuli have been observed to occur at predictable time points post stimulus onset (Coles & Rugg, 1996). Thus, in cases where the time of the stimulus appearance is known, a classifier can hope to take advantage of these so-called *event-related potentials* (ERPs) by attempting to classify samples which are time-locked to the onset of the stimulus. For this reason, I used the time-locked, rather than the steady-state, approach for most of my analysis.

7.1.3 EEG Classification

EEG data is extremely noisy, which makes classifying it correspondingly difficult. Some sources of noise are external. These include the aforementioned motion artifacts and noise introduced by the EEG collection device. One controllable source of external noise is the alternating current in the electric mains (Newman E. , 2008). Conveniently, Emotiv filters out frequencies between 50 and 60Hz in an attempt to improve signal quality (Emotiv, 2010). However, EEG is also plagued with internal noise from various brain processes. To make matters worse, ERP amplitudes are typically small, meaning that the responses can easily be drowned out by noise (Coles & Rugg, 1996). For this reason, researchers looking to detect ERPs typically average hundreds of trials to reveal a general pattern which is otherwise effectively invisible (Coles & Rugg, 1996). Of course, a classification algorithm does not have this luxury, since each unknown sample must be classified individually. To be successful in classification, then, it is often valuable to perform various preprocessing steps which improve the signal-to-noise ratio (SNR) of the data before invoking a machine-learning algorithm.

7.1.4 Online Classification

Performing classification in an online manner is more difficult than offline classification for several reasons. First, the classifier must train relatively quickly. While this is not an issue for many learning algorithms, it may rule out some procedures, such as training a classifier many times on different feature sets or with different random seeds (in the case of an algorithm with a random element), which are used in offline studies (Newman E. , 2008). Round-based learning algorithms, such as AdaBoost and VotedPerceptron (see 7.2.5 and 7.2.4) are useful in time-sensitive scenarios because their training time can be manually configured. However, choosing a smaller number of rounds for the purposes of faster training may compromise accuracy to some degree.

It is also important to consider how the performance of offline and online classifiers is validated. A

popular offline means of assessing classifier performance is leave-one-out or k-fold validation (Newman E. , 2008) (Parra, Spence, Gerson, & Sajda, 2005). In such procedures, the classifier always has the benefit of using a high percentage of the total available data when making its predictions. In contrast, an online classifier may only be able to use a smaller percentage of the total data. Even when the classifier is periodically retrained, early online classifications still have significantly less data to work with than would offline classifications performed on the total data set. Finally, an experiment which allows the subject to make choices and which reports classifier progress along the way must almost necessarily present stimuli at a slower rate than one in which the subject is a passive observer (at the very least, this holds true in Experimenter). Since classifiers typically benefit from an increased volume of data, an online classifier is at an additional disadvantage: the total number of samples it has to work with will almost certainly be smaller than an offline classifier processing data from a similar-length experiment.

7.1.5 Voltage-based Preprocessing

The most straight-forward approach to EEG analysis is to use the raw signal voltages directly. Since the collected data is never perfectly aligned with the onset of the stimulus (both because the data resolution is finite and because the combination of timers and multithreading means that exact presentation time cannot be guaranteed), it is often helpful to down-sample the time series so that each data sample has the same sequence of time bins. This also helps reduce the effect of very high frequency noise. I typically used down-sampling to reduce the resolution to 20ms instead of the native 7.8ms; this does not noticeably alter the overall shape of the time series. Down-sampling also reduces the dimensionality of the data, which is very important for some classifiers and nearly always decreases training time.

Another common preprocessing step is to subtract the sample mean from each channel time series (effectively setting the mean to zero). The goal of this is to reduce the impact of low-frequency drift in classification. Of course, this also risks the possibility of removing useful information; perhaps some stimuli induce a general increase or decrease in voltage along a channel. One solution to this problem is to expand the sample to include some amount of time before and after the onset of the stimulus for the purposes of calculating the mean (Newman E. , 2008). Assuming that the before and after periods are neutral, this should retain useful information while still removing low-frequency drift. Of course, this may still be insufficient since in my experiments subjects know in advance which type of stimulus they will be viewing;

this anticipation may also affect the average voltage. Furthermore, in side-by-side presentation experiments the subject is likely to move before the trial begins in order to focus on one side of the screen or the other. To avoid having a motion artifact skew the average, it might be necessary to make the fixation period substantially longer. Another possibility is to subtract off the channel mean but to also include this mean value as a feature during classification. This allows the machine learning algorithm to determine whether the sample mean is an informative feature or simply random noise.

7.1.6 Time-Frequency Analysis

A classic approach to signal processing is to transform the signal from the time domain to the frequency domain. The frequency domain representation of a signal can prove more tractable to analysis than the time domain representation both because signals are often well-characterized by one or more frequencies (e. g. musical notes) and because sources of noise whose frequencies are different than the relevant frequencies in the signal (e. g. the electric mains) will be effectively separated out by the transformation. In working with EEG, the goal of frequency analysis is typically to determine the signal's power spectrum, that is, the power at each frequency. This can be computed via the (discrete) Fourier transform, which breaks a signal into its constituent frequencies. However, some signals, such as EEG, have frequency domain representations that change over time. In these cases, an approach such as the short-time Fourier transform or a wavelet transform can be used to capture both time and frequency information (Newman E. , 2008).

Over the course of the project, I tried using both discrete Fourier transform- and wavelet-based preprocessing to improve the performance of my classifiers. In general, I found that use of the Fourier transform did not improve classification. Wavelets seemed to be a more promising approach, but here my progress was stymied by the lack of available wavelet libraries for .NET. The one implementation I did find was extremely limited in that it did not allow the frequencies of the wavelets to be specified and only processed a small number of frequencies. Applying this transform to the sample time series also failed to improve classifier performance (in fact, performance typically decreased).

7.1.7 Linear Discriminant Analysis

Parra, Spence, Gerson, and Sajda suggest that, due to the Ohmic nature of tissue, EEG activity can be accurately described with a linear forwarding model that relates current sources within the brain to

surface potentials (Parra, Spence, Gerson, & Sajda, 2005). This fact is used to justify an approach based on linear discriminant analysis (Parra, Spence, Gerson, & Sajda, 2005). Essentially, the goal to find a weight vector w which can be used to project the multidimensional (multichannel) EEG data at each time point onto one dimension, thus leaving only a single combined channel whose features depend on the choice of w (Parra, Spence, Gerson, & Sajda, 2005). Parra et. al. describe several methods of choosing a w so as to maximize the evoked response difference between EEG activity observed under two different experimental conditions, including an approach based on trial averaging and a more sophisticated one using logistic regression (Parra, Spence, Gerson, & Sajda, 2005). The average output of projecting each time point's worth of data can then be used to classify a trial (Parra, Spence, Gerson, & Sajda, 2005).

I found that this method was insufficiently powerful to classify the data collected with the Emotiv. This is likely due to Emotiv's lack of electrodes, since the channel count is equal to the number of features available to the discriminator (the data used in Parra et. al.'s study had 64 channels) (Parra, Spence, Gerson, & Sajda, 2005). I also attempted a modified approach, which was to train a separate discriminator for each time bin. This allows the classifier to take advantage of the fact that the relationship between channels is not constant over time. On the other hand, it also greatly reduces the size of the training set available to each discriminator. For the data I collected, this modification improved classification, but the results were still sub-par. However, this technique inspired another more successful one which will be described later (see 7.2.6).

7.2 Classification Algorithms

Since Experimenter supports online classification, it was vital that the application have access to machine learning algorithms from the .NET framework. One disadvantage of working with the .NET platform is the lack of open-source libraries providing access to implementations of machine learning algorithms. I considered using the .NET NPatternRecognizer library, but ultimately decided against it because of the clumsy APIs and messy, incoherent source code (which is commented in Chinese and contains a great deal of dead/commented out code) (Liu, 2009). Another option I considered was to use IKVM, a tool which ports Java binaries to the .NET runtime along with the Java WEKA machine learning library (Frijters, 2010) (University of Waikato, 2011). However, doing this requires taking dependencies on ported versions of much of the Java standard library, which significantly increases the application's

footprint. Furthermore, when I tried IKVM with an existing Java application I found that it not only ran slowly but also that there seemed to be bugs in some of the standard library ports. Finally, I considered using the native libsvm library, possibly via .NET's P/Invoke functionality which allows managed code to call native code (Chang & Lin, 2011). Here I was stymied by a lack of documentation as well as a concern about the emergence of P/Invoke errors, which are notoriously difficult to debug. Indeed, any solution that involved calling code outside the .NET framework sacrificed the ability to use Visual Studio's excellent debugger. Thus, I ultimately decided simply to implement several classification algorithms on my own. These algorithms are summarized here.

7.2.1 K-Nearest Neighbors (k-NN)

The k-nearest neighbors algorithm is a simple classifier which operates by finding the k examples in the training set which are closest to the test example by some metric (e. g. Euclidean distance). A prediction is then computed as the most common class among this set of k nearest neighbors, with each neighbor's vote assigned a weight inversely proportional to that neighbor's distance from the test example. The naive implementation of this algorithm has the advantage of being extremely quick to train, since training consists only in saving a copy of the training set. I implemented this algorithm primarily because it was used by Lillian Zhou in her previous work with Emotiv (Zhou, 2010). However, I found the algorithm to be only moderately effective when dealing with EEG data. In Experimenter, the configurable parameters of the k-nearest neighbors algorithm are:

- ◆ *K*: the number of nearest neighbors used in the voting process.
- ◆ *WeightedVoting*: a Boolean value which determines whether or not all votes are weighted by the voter's Euclidian distance to the test example (as opposed to being given equal weight).

7.2.2 Penalized Logistic Regression (PLR)

Logistic regression is a classification technique that seeks to fit the training set to a logistic curve by projecting it onto one dimension and passing the result to a logistic function (Parra, Spence, Gerson, & Sajda, 2005). This is done by deriving an optimal vector of weights used to perform the projection (Parra, Spence, Gerson, & Sajda, 2005). Penalized logistic regression (PLR) looks to improve on this technique by adding a penalty term that discourages weights from becoming too large, which can happen with perfectly separable classes and may be a cause of over-fitting (Parra, Spence, Gerson, & Sajda, 2005). My

implementation of PLR is based on an iteratively reweighted least squares algorithm described in Parra et. al. (Parra, Spence, Gerson, & Sajda, 2005). In Experimenter, the configurable parameters of the PLR algorithm are:

- ❖ λ : the penalty term.
- ❖ *MinDistance*: the epsilon value used to determine when the weights have converged in the iterative algorithm.
- ❖ *MaxIterations*: an optional upper bound on the number of iterations performed.

7.2.3 Decision Stump

The decision stump is an extremely simple classifier which uses only a single feature of the data. In the training stage, the algorithm chooses a cut-point. Classification of test examples is determined by testing whether the selected feature in the example falls above or below the cut-point (Russel & Norvig, 2003). The cut-point itself is chosen so as to minimize classification error on the training set. Although the decision stump is a rather weak classifier, one advantage of it is that it can handle cases where examples in the training set do not all have equal weight. This allows decision stump to be used as a so-called ❖weak learner❖ in the AdaBoost algorithm (described below). In Experimenter, the configurable parameter of the decision stump algorithm is:

- ❖ *Feature*: the index of the feature to be used by the algorithm.

7.2.4 Voted Perceptron

The voted perceptron algorithm is conceptually similar to more complex support vector machine algorithms in that it works by (possibly) mapping the data into a very high dimensional space and then locating the hyperplane which separates the two classes and which maximizes the distance to the nearest training data points (Freund & Schapire, 1999).❖ On the surface, this requires that the data be linearly separable. However, the ❖kernel trick❖ allows simple functions to be used to map the data into even higher dimensions where it may *become* linearly separable (Freund & Schapire, 1999). To support this feature, I implemented several popular kernel functions to go along with the voted perceptron algorithm, including the basic linear kernel $(x \cdot y)$ ❖and the polynomial kernel $(scale * (x \cdot y) + offset)^{exponent}$. Other advantages of the algorithm include its speed, its ability to learn from new examples in an online manner, and its ability to incorporate example weights, much like the decision stump. In Experimenter, the

configurable parameters of the voted perceptron algorithm are:

- ❖ *Epochs*: the number of iterations performed over the training set.
- ❖ *Kernel*: the kernel function used by the algorithm. Individual kernels may have their own configurable parameters.

7.2.5 AdaBoost

AdaBoost is an ensemble learning algorithm: it derives its predictions by intelligently combining the predictions of a group of ❖weak learners❖, other classifiers trained as part of the AdaBoost training process (Schapire, 2003). As each new weak learner is trained, the algorithm adjusts the weights of the examples in the training set to reflect the success or failure of the current set of weak learners in classifying that example. These weights are then used in training future weak learners, biasing them towards trying to classify more difficult examples. In Experimenter, the configurable parameters of the AdaBoost algorithm are:

- ❖ *Rounds*: the number of weak learners trained by the algorithm.
- ❖ *WeakLearnerTemplate*: the type of classifier to be used as a weak learner. This classifier must be able to work with arbitrarily weighted training examples.
- ❖ *WeakLearnerTrainingMode*: this setting determines which subset of features is used in training each weak learner. For example, the *AllFeatures* setting trains each weak learner using every feature of the data, while the *RandomFeature* setting trains each weak learner on a single random feature of the data (this is useful when the weak learner is a decision stump).

7.2.6 AdaBoost and Voted Perceptron

When classifying a time series using their linear discriminant analysis approach to classification, Parra et. al. simply average the output of the discriminator at for all time points in the series in order to formulate a prediction for the entire series (Parra, Spence, Gerson, & Sajda, 2005). This approach does not take into account the fact that some time points may be more informative in terms of discriminative ability than others. Thus, it seems reasonable to combine the discriminator predictions in a more ❖intelligent❖ manner. The AdaBoost algorithm offers one means of doing this. If we view the discriminator trained on each time point as a weak learner, then AdaBoost could be used to guide the training of the discriminators in order to combine their outputs for an effective overall prediction. Of course, this procedure will not work

with decision stumps, since in order to project 14 channels onto one the weak learner must be able to work with multiple features. For this reason, I experimented with using AdaBoost with the voted perceptron algorithm as a weak learner. Rather than training the weak learner on the full data set or, as with decision stumps, on a single feature, the algorithm trained each weak learner on one time bin's worth of data (14 features). This resulted in a fairly successful classifier.

8 Results

This section details the numerical results obtained from various experiments. Aside from the eyes open vs. eyes closed experiment where over 90% accuracy was achieved, most of my successful results fell within the 60-70% range. While such results may not seem impressive it is important to consider them in the context of the noisy, 14-channel data stream from which they were obtained as well as the complexity of the underlying mental states that these experiments attempted to differentiate.

8.1 Eyes Open vs. Eyes Closed

Since the expected EEG response to the eyes closed condition is a steady state response rather than a time-sensitive ERP, voltage-based classification that compares the voltages at particular time bins under each condition is generally ineffective despite the high separability of the two conditions. For this reason, the average amplitude of the signal (approximated by the average difference from the signal mean), rather than the voltage time series, was used as input to the classifier. Furthermore, only one occipital channel's worth of data was used (either O1 or O2). A simple decision-stump classifier trained on this single-channel amplitude proved very effective (> 90% accuracy) at differentiating between the two conditions online even with very few (10 - 20) training trials. Most classifier failures could be attributed to eye-motion artifacts. For example, sometimes holding one's eyes open causes them to feel very dry, leading to a fluttering of the eyelids or even a blink. Such artifacts can sufficiently distort the average amplitude to fool the simple classifier.

Two easy steps might be able to improve the classifier performance on this task to very near 100%. First, tuning the artifact detection algorithm to prevent its triggering during eyes closed trials would improve accuracy by allowing it to detect and eliminate most, if not all, erroneously classified trials (the above results were obtained with artifact detection disabled to avoid otherwise frequent false positives). Another

option would be to classify based on the power spectrum of the occipital signals rather than the amplitudes, since this would preserve more discriminatory information and would likely be more robust to motion artifacts. These improvements were not pursued, however, since the current performance is indicative of Emotiv/Experimenter's ability to perform very well on this task.

8.2 Offline Validation

The offline accuracy results described in the subsequent sections were computed in the following manner:

1. Reject all trials with motion artifacts using the default motion-artifact detection parameters. This was done because in some cases the experiments were run without artifact detection enabled.
2. For each trial, down-sample the time series in that trial to a resolution of 20ms.
3. For each channel in each trial, subtract the channel mean from each data point. Append this channel mean to the resultant series.
4. Concatenate the feature sets from each channel into a single feature set.
5. For each data set, partition examples into two collections, one for each class.
6. Randomly shuffle each collection of examples.
7. Reserve the first 10% of the examples in the first collection for the test set. Reserve the *same number* of examples from the other collection for the test set as well. Thus, chance performance of a classifier on the test set is always $\frac{1}{2}$.
8. Concatenate and then shuffle the remaining examples to form the training set.
9. Train each classifier on the training set.
10. Record each classifier's accuracy in classifying the test set.
11. Repeat steps 6-10 100 times, and recording the average accuracy of each classifier.

Note that this process is similar to ten-fold validation except that it runs ten times as many tests and is randomized. This process was performed using the following classifiers:

◆ *AdaBoost VP:*

o *Algorithm:* AdaBoost

- o *Rounds*: the number of features per channel
- o *WeakLearnerTrainingMode*: each weak learner was trained on a random time bin's worth of data where one bin actually consisted of the 14 channel means.
- o *WeakLearnerTemplate*: Voted Perceptron
 - ◆ *Epochs*: 4
 - ◆ *Kernel*: Polynomial with degree 3

◆ *AdaBoost DS*:

- o *Algorithm*: AdaBoost
- o *Rounds*: the total number of features
- o *WeakLearnerTrainingMode*: each weak learner was trained on a single random feature
- o *WeakLearnerTemplate*: Decision Stump

◆ *PLR*

- o *Algorithm*: Penalized Logistic Regression
- o λ : 10^{-4}
- o *MaxIterations*: 50
- o *MinDistance*: 10^{-5}

◆ *KNN*

- o *Algorithm*: K-Nearest Neighbors
- o *K*: 20
- o *WeightedVoting*: Yes

These classifier parameters were chosen based on my experience working with these classifiers over the course of the project. A thorough parameter scan could surely improve performance of at least some of the classifiers at the cost of a great deal of computation time.

8.3 *Faces vs. Places*

8.3.1 **Version 1**

The following chart (Figure 1) shows the performance of the four classifiers on three version 1 data

sets. The first set is from an experiment where the two stimulus classes were faces and manmade objects, while the second and third sets used faces and outdoor scenes.

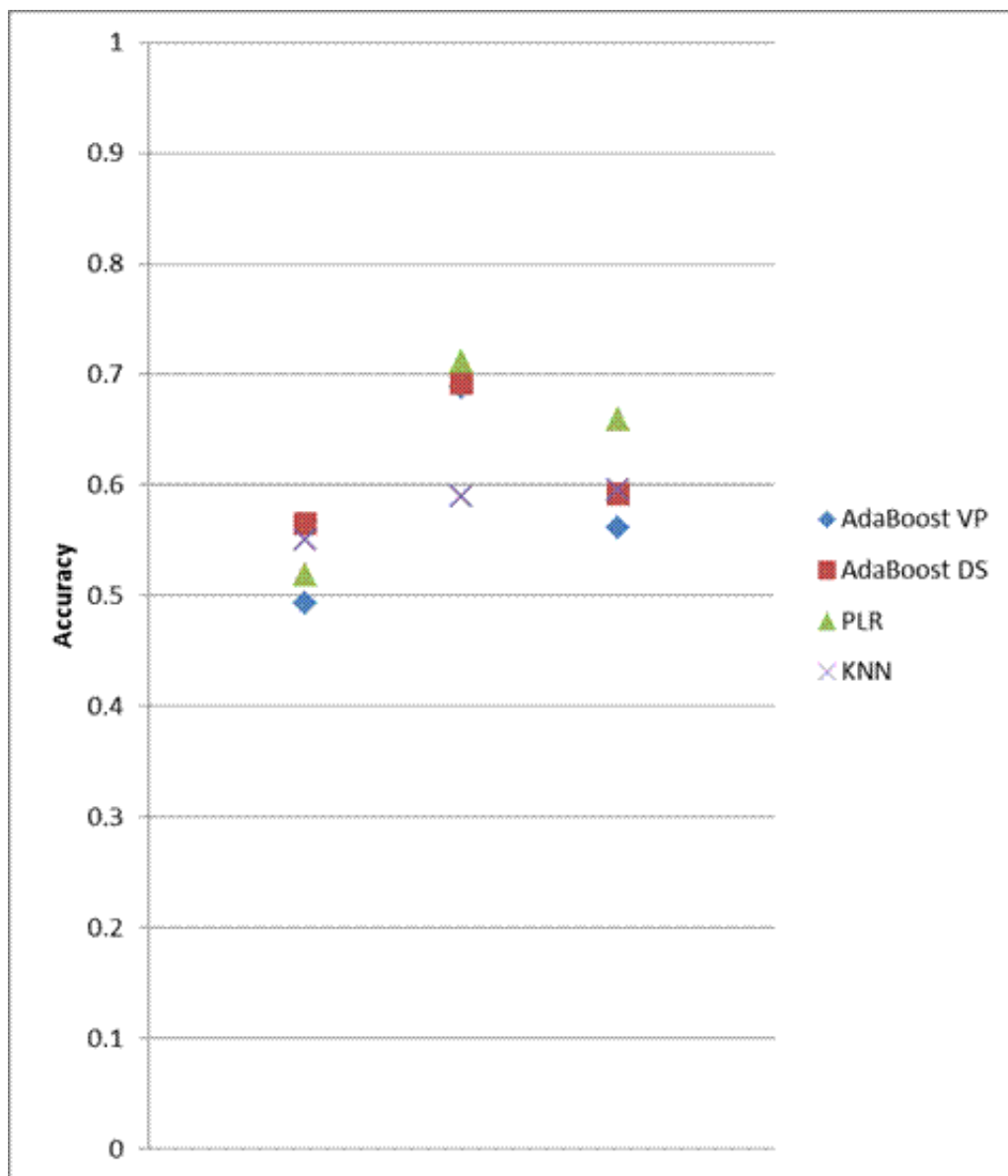


Figure 1 Version 1 results

Clearly, the face vs. scenes data classified significantly better than the faces vs. objects data. This could be a fluke, or it could be indicative of the fact that faces vs. scenes is not equivalent to faces vs. non-faces (i. e. the particular choice of places matters as well). For the faces vs. scenes data sets, the PLR classifier had the best average performance.

8.3.2 Version 2

Figure 2 shows the poor classifier performance on data generated in the superimposed version of the faces vs. places experiment. This is consistent with the observed online results which led to the creation of the side-by-side version of the experiment.

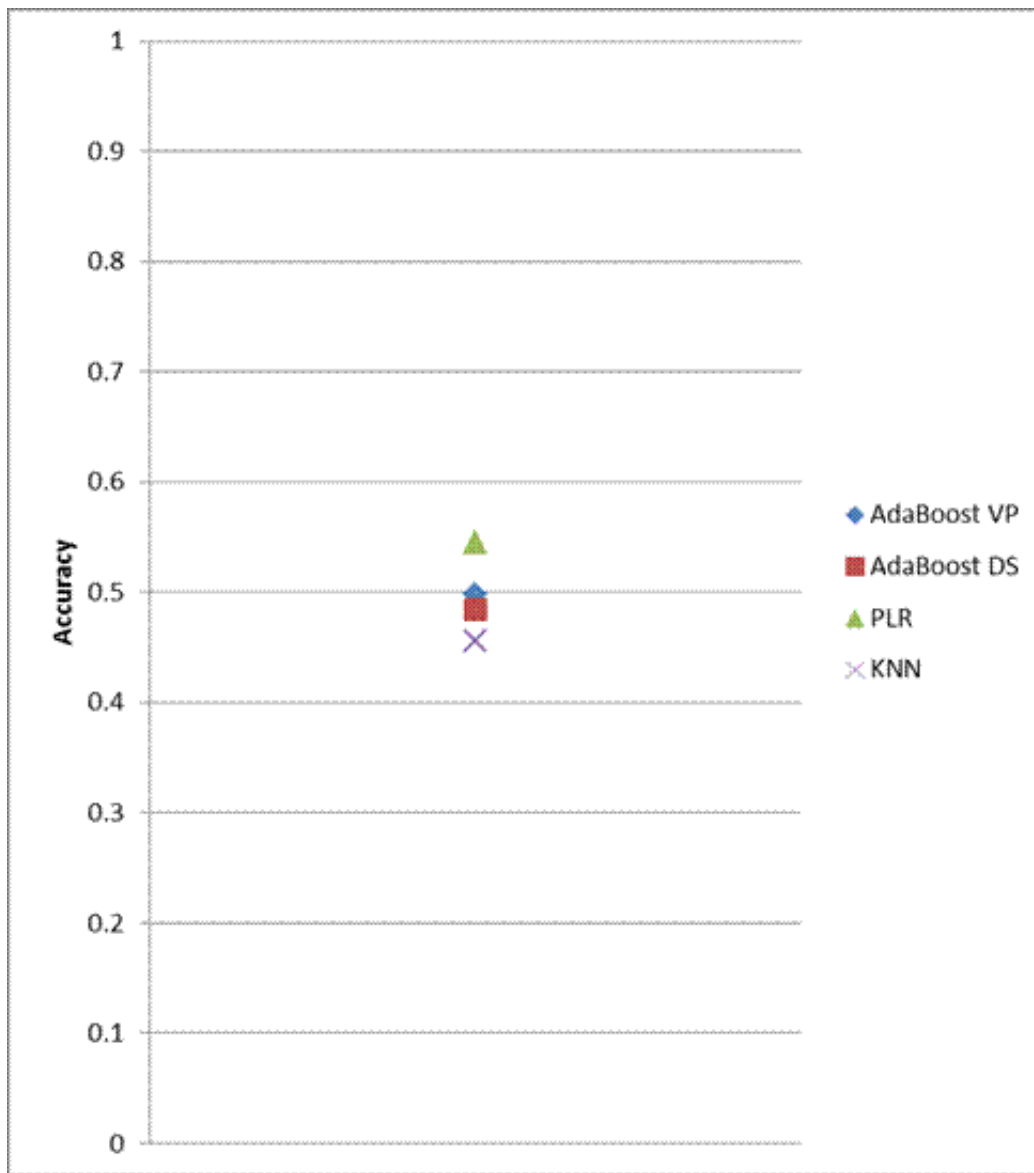


Figure 2 Version 2 results

8.3.3 Version 3

Figure 3 shows the performance data for a number of data sets collected using version 3 of the experiment with myself and one other subject.

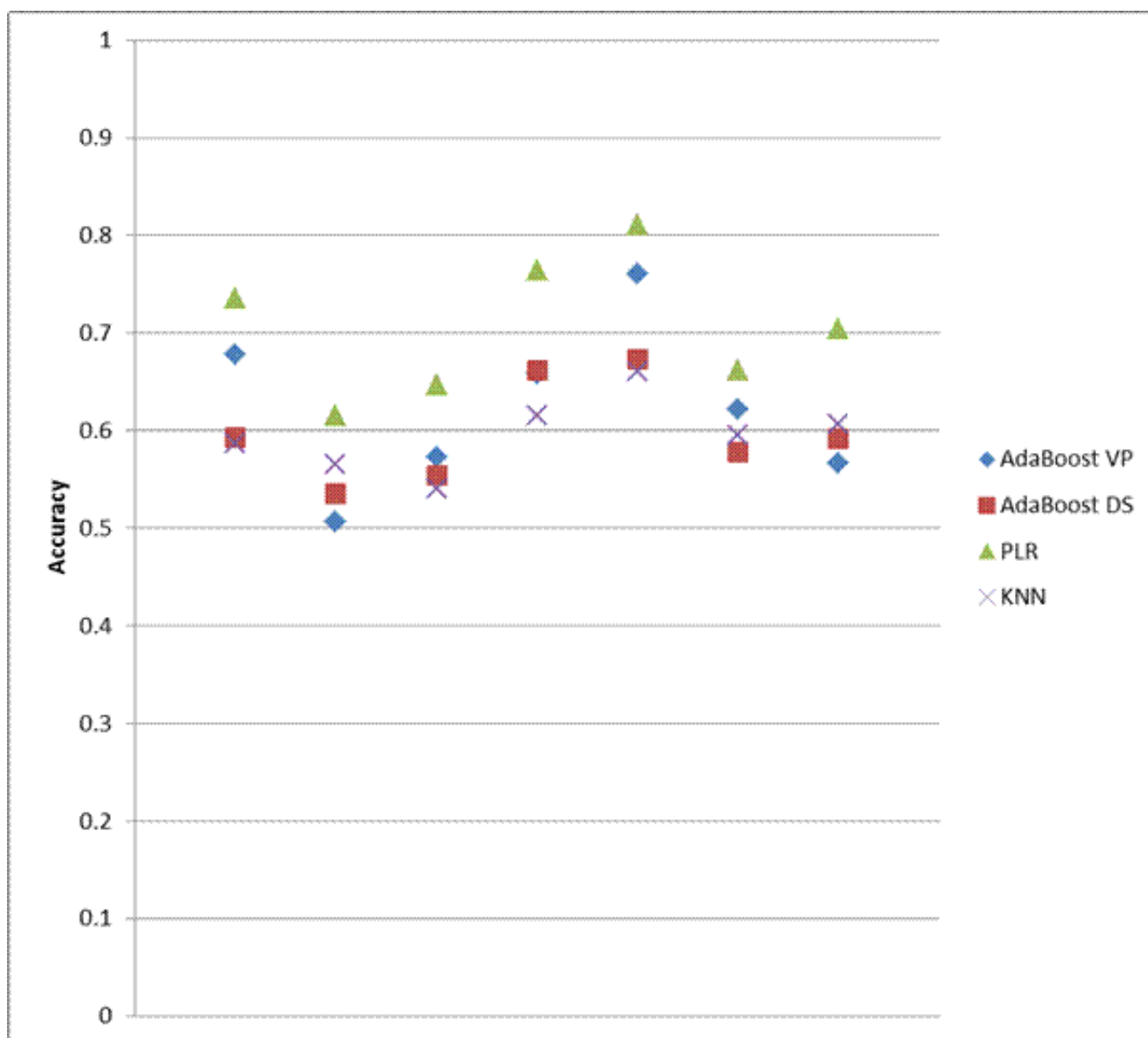


Figure 3 Version 3 offline results

Many of these data sets classify very well, with the PLR classifier breaking the 80% mark on one set and the 70% mark on several others. The data set that achieved the best performance was one where I experimented with using a smaller image size than normal (200x200 instead of 400x400). This makes it even easier for the subject to avoid moving his or her eyes. The observed online performance results for this experiment were slightly worse than the offline results (60-66%), although typically the online classifiers used in these experiments were not as well-tuned as the classifiers used in the offline analysis.

◆◆◆◆◆◆◆◆◆◆ To better understand online performance, then, I simulated online classification with a periodic retraining every ten trials for each data set. Thus, the classifiers were trained on the first ten trials and judged on their ability to predict trials 11 through 20, then trained on the first 20 trials and judged on their predictions for trials 21 through 30 and so on. The accuracy score at each trial presented here is a

running score taking all predictions so far into account (this is the same score displayed by Experimenter).

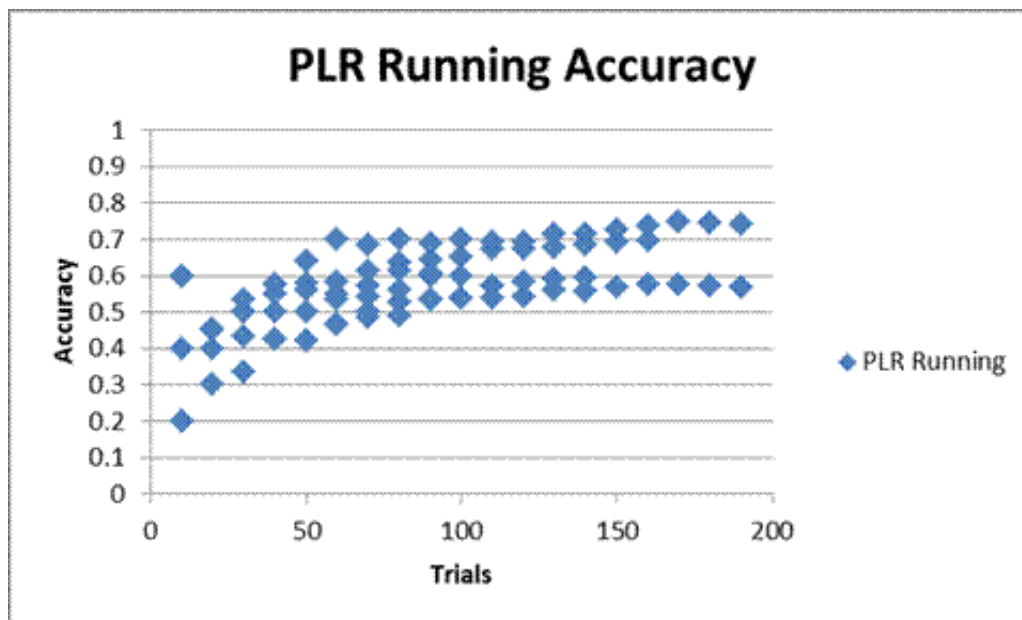


Figure 4 Version 3 PLR online accuracy

The result of this simulation was not surprising. Accuracy steadily improved before eventually leveling off at a value comparable to that found in the offline performance tests (Figure 4, see Figure 26, Figure 27, and Figure 28 for the results for the other classifiers). The one exception to this trend was the KNN classifier, which maintained roughly constant accuracy throughout on most data sets. For the other classifiers, though, these results unfortunately suggest that nearly 100 training trials are necessary before online classifier performance reaches its peak. While such experiments could be run, in many cases it seems worthwhile to sacrifice some online performance to reduce the number of training trials, creating a more enjoyable experience for the subject (since the mind reading aspect of the experiment only comes online in the test phase).

8.4 Artist vs. Function

Despite a preliminary online accuracy result of over 60% using a classifier very similar to AdaBoost VP, in general the data collected during the artist vs. function experiment proved rather resistant to classification. Figure 5 shows three attempts to classify artist vs. function data sets from two subjects using different feature-selection techniques:

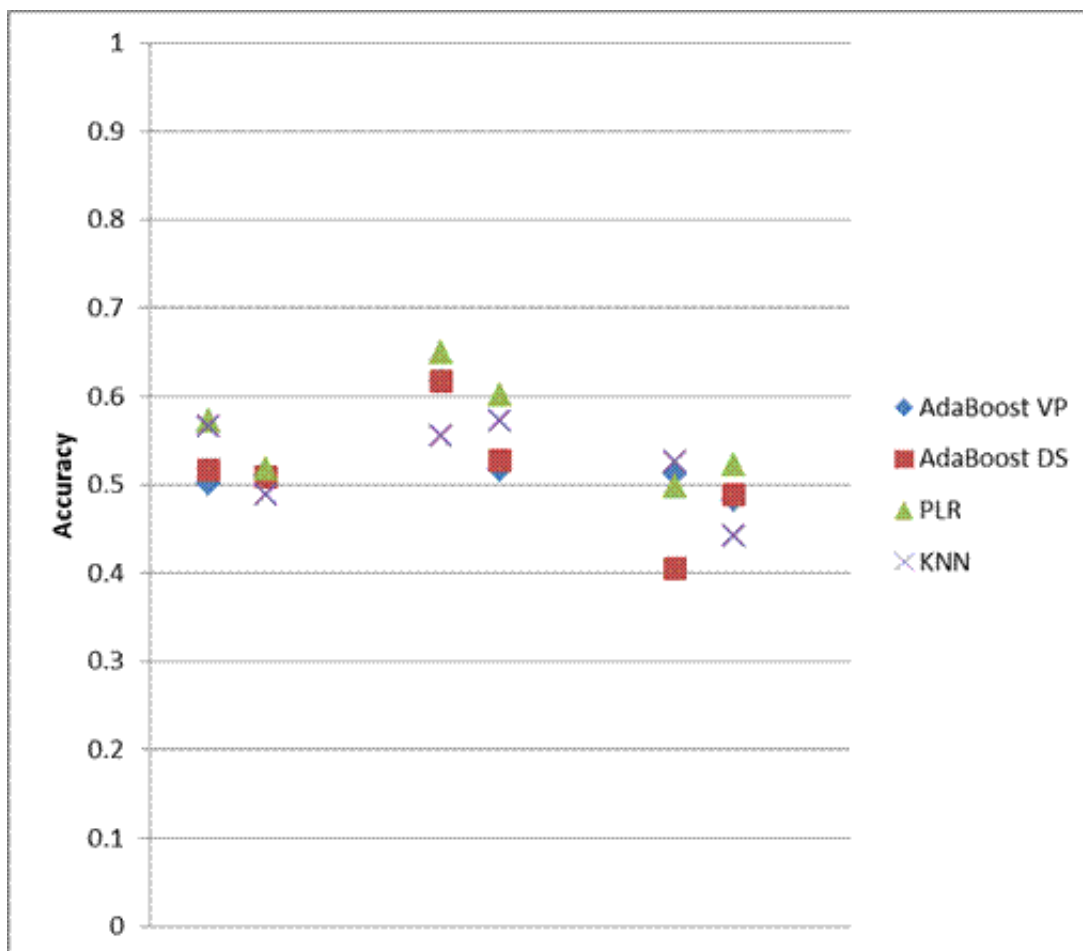


Figure 5 Artist vs. function results

The first pair of results used the standard feature selection routine discussed above, but explicitly took only the first 500ms of each trial (this truncation was done to keep classifier training times reasonable, especially for PLR). All classifiers performed poorly under these conditions. The second pair of results used only the second 500ms of each trial. This was motivated by the fact that both tasks require a willful action to initiate, which likely causes the response to be slower than for something more automatic like image processing. Whether or not this is the case, the data from this later time period proved more classifiable, resulting in moderately good performance (especially on the first data set). Finally, the third pair of results was obtained by training the classifiers on the power spectrum (as computed by the FFT) of each trial. This was attempted because the subject's performing the artist or function task seemed more like a steady-state phenomenon than a time-sensitive response to a stimulus. However, the poor classifier performance under these conditions suggests either that this is not true or that more sophisticated time-frequency analysis methods are required.

8.5 Faces vs. Expressions

As with the artist vs. function experiment, I was able to obtain some relatively good online results (60-65% accuracy) for this experiment, but found classifying data collected for offline analysis to be very difficult. Figure 6 shows three attempts to classify faces vs. expressions data sets from myself and another subject using the previously described trio of feature selection techniques (except that rather than using the 500-1000ms time period for the second pair of results I used the 200-700ms time period):

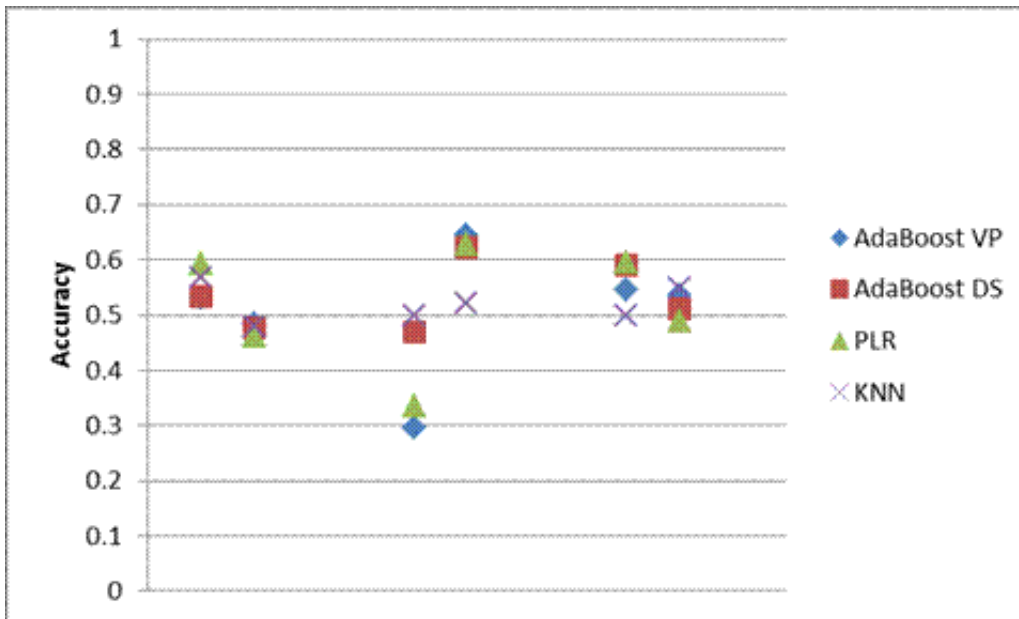


Figure 6 Faces vs. expressions results

In general, classification was mediocre at best under all conditions. Furthermore, unlike with the artist vs. function data, the two data sets seem to behave differently from each other under the various feature-selection regimes.

8.6 Overview

The results described above demonstrate that the data collected with Experimenter can often be effectively classified, particularly in the eyes open vs. eyes closed and the faces vs. places cases. The data from the artist vs. function and faces vs. expressions experiments proved more difficult to classify, although this may be partially attributable to the fact that, due to time constraints, I only ran a few instances of each experiment, granting little opportunity either to tune the experiment or to train the subjects (unlike the faces vs. places experiment, these experiments require a degree of mental discipline).

Of the machine learning algorithms considered, PLR proved to be the most reliably effective at classifying Experimenter data (at least given the current set of classifier parameters). However, PLR was

also by far the slowest classifier to train, taking five to ten seconds with 200 examples and 500ms worth of data (25 features/channel after down-sampling). The other algorithms generally took less than one second under the same conditions. This is especially troubling when using PLR for online classification.

Interestingly PLR's training time is generally dominated not by the size of the training set but by its dimensionality (at least with fewer than 1000 examples). The algorithm I implemented must invert an $n \times n$ matrix on each iteration, where n is one greater than the example dimension. This can be a problem, since Emotiv's EEG data has hundreds of features even for a short trial. For this reason, most of the algorithm's training time is spent inside calls to Math.NET's matrix inversion routine. As the number of features increases (perhaps from using longer trials), the PLR algorithm quickly becomes effectively unusable. Thus, to use PLR one must be more careful to reduce the size of the feature set than with the other algorithms. This can be done through trial and error, through leveraging experiment-specific neurological knowledge (e. g. the time of a known ERP), or through increased down-sampling.

9 Discussion

9.1 Using the Emotiv Headset

The development of Experimenter necessitated substantial use of the Emotiv headset. This experience allows me to speak to the product's usefulness as a BCI and EEG research device. While Emotiv's low cost, portability, and relative ease of use are obviously exciting in that they simplify the process of executing experiments, perhaps what will prove even more useful to researchers is the device's potential to greatly expedite the experimental design and development process. With Emotiv, experiment designers can easily test their designs on themselves and collaborators long before running an expensive formal experiment. It would also be interesting to see researchers collaborate with Emotiv and Emotiv application developers to collect data on a larger scale. For example, Emotiv applications could give users the option of logging their EEG activity and periodically uploading it to a central database where it could be anonymized and made accessible to researchers (many commercial applications already upload usage data in this way for product improvement purposes). Such data would obviously be of lower quality than that collected under controlled conditions in a laboratory, but presumably its sheer volume and low cost would make it of great interest nonetheless. Of course, implementing such a scheme would require Emotiv

to at least partially relax its restrictions on making raw data available to applications (since this is currently available only through the research edition). Presumably though, this feature could be implemented so as to maintain Emotiv's hold on the real-time raw data stream and/or so as to provide Emotiv with a financial incentive (researchers might pay for access to the data). At the very least, one would assume that Emotiv might wish to use the raw data collected in this way to improve its own classification suites.

Despite the many exciting possibilities, though, there are some distinct pitfalls one can encounter when the Emotiv headset. One issue is that the incoming data seems to become increasingly noisy as the battery nears empty. This is particularly frustrating since the battery meter in the Emotiv TestBench application does not seem to work properly and thus the problem can easily go unnoticed. For researchers hoping to collect valid data using Emotiv, this means that it is necessary to recharge the battery frequently to avoid corruption. Also, although setting up Emotiv is obviously far easier than for a traditional EEG setup, properly aligning the device so as to get good connectivity on all electrodes can be difficult and frustrating, especially for new users. Practice with the headset also seems to be important for obtaining the best experimental results; in general I observed that myself and the one other subject whom I worked with most tended to produce more classifiable data than did other subjects. This is likely caused by acclimation to the experimental setup. A user's learning to maintain both focus and a consistent cognitive approach to each experimental task (e. g. evaluating a mathematical expression) during the experiment likely reduces the noise present in the collected data.

9.2 *Future Directions*

9.2.1 **Experiments**

There are a multitude of possible experiments which one could attempt using Emotiv and Experimenter. For example, one might try to distinguish between a subject's response to other classes of stimuli, such as emotional and unemotional images. Another classic paradigm compares the subject's response to familiar versus oddball images. This could be implemented by creating one stimulus class with a small number of images and another with a large number of images. A variety of two-task experiments similar to the artist versus function experiment are also possible. Finally, the freedom of motion creates the possibility of developing experiments where the conditions do not solely depend on looking at the screen, but instead attempt to detect the user's engagement in some other sort of activity.

9.2.2 Experimenter Functionality

The Experimenter application could be improved by adding a greater degree of configurability to the experiment setup and classification suites. For example, the experiment could be made more general by supporting a wider variety of tasks (currently only binary choice tasks are supported). The ability to input arbitrary integers, for example, would allow Experimenter to use a variety of counting and calculation tasks. Experiment designers may also wish to be able to directly manipulate probabilities used in the experiment. In particular, the oddball experiment would work best if the oddball category of images could be made to appear less frequently.

Increasing the number of available preprocessing options would be a good first step towards improving Experimenter's online classification capabilities. For example, it would be beneficial if users could select from a variety of additional data features including channel amplitudes and time-frequency representations when constructing the input to a classifier.

Integrating Python with Experimenter would do much to improve the application's data-processing capabilities. IronPython is an open source implementation of the popular Python programming language that is fully integrated with the .NET framework (Microsoft, 2011). Use of IronPython would give Experimenter access to powerful analysis libraries such as the popular PyMVPA library (Hanke, Halchenko, Sederberg, Hanson, Haxby, & Pollmann, 2009). Another great benefit of embedding Python would be the ability to support arbitrary preprocessing via inserting a user's Python script into the preprocessing tool chain at runtime (the script could be typed/copied directly into the Experimenter GUI). While this is also possible using dynamically compiled C# code, those looking to minimize programming effort are likely to find Python's syntax easier to work with.

9.2.3 New Projects

There are also many interesting possibilities when one considers building other sorts of applications based on Experimenter's code base for interacting with the Emotiv headset and with Emotiv data. These capabilities, along with Experimenter's flexible presentation system, provide programmers with useful tools for rapidly developing Emotiv-based applications which are very different in nature than Experimenter. To demonstrate this concept, I constructed two simple applications which use Experimenter's core code in slightly different ways. Using Experimenter's presentation, classification,

and Emotiv interoperation suites, I was able to build each of these applications in about half an hour with roughly 100 lines of C# code.

9.2.3.1 *Image Classifier*

A great deal of work has gone into developing algorithms for classifying images (e. g. face detection), something which the human brain naturally excels at. Thus, I thought it would be interesting to build an application whose purpose was to categorize images based on a viewer's EEG response. The application begins with a stimulus class containing two categories of images, such as faces and scenes. Only a few images of each type have been categorized; the rest are unmarked. The application then begins displaying the classified images to the user in rapid succession (400ms each) until each image has been displayed ten times. The collected responses to each image are then averaged, and a classifier is trained on the resultant trial-averaged signals. Then, the application repeats this process with the unclassified images. Whenever ten artifact-free responses are collected for a single image, the application attempts to classify that image based on the trial averaged response. It then retrains the classifier, including the newly classified image in the training set (the user either confirms or denies the correctness of the classifier's prediction). This continues until all images are classified. When I tested the image classifier using an AdaBoost/voted perceptron classifier, the application achieved 78% accuracy while classifying 42 face and scene images, beginning with only 5 marked images of each type.

9.2.3.2 *Thought-meter*

Unlike Experimenter and the Image Classifier, which rely on time-locked data to perform classification, the Thought-meter explores the possibility of steady-state classification. This application first attempts to learn to distinguish between a neutral condition and some user-chosen condition (e. g. thinking about a specific topic) by recording 20 seconds of data under each condition and before training a classifier. The application then displays a counter to the user which behaves in the following manner: every time new data is received from the Emotiv, the last 500ms of data are fed to the classifier. When the user-defined condition is detected, the counter increases by one, and when neutral condition is detected, the counter drops by 1.5 to a minimum of zero. The user can then attempt to move the counter away from zero (which is difficult due to the bias towards neutral) using his or her mind. A test found the thought meter to

succeed in differentiating between watching a movie and staring at a blank screen as well as between steady-state smiling (steady-state because the initial action of smiling triggers the artifact detection algorithm) and holding a neutral expression. However, I had little success with getting the current implementation to differentiate between purely cognitive conditions.

9.3 Conclusion

This work resulted in a functioning, full-featured, and well-documented application which is capable of implementing a variety of interesting experiments using the Emotiv headset. The application supports both data collection for offline analysis as well as exciting online classification capabilities. These abilities were tuned through an exploration of various preprocessing and machine-learning algorithms, the results of which are presented here.

Another benefit of this work to future students and researchers is its development of a robust, high-level API for interacting with the Emotiv headset. This API lowers the effort bar for building interesting applications that work with raw EEG data, rather than just with Emotiv's built-in classification suites. By using an open classification and data processing framework rather than limiting themselves to Emotiv's classification APIs, developers could benefit from the substantial knowledge and experience of the neuroscience community while also creating applications that could be used to further our understanding of the brain.

10 Acknowledgements

I would like to specially thank several people who contributed to the success of this project:

- ❖ Lillian Zhou, for providing me with all of her data, code, and write-ups from her previous independent work with Emotiv.
- ❖ Jarrod Lewis-Peacock, for providing me with a huge number of stimulus images for my experiments.
- ❖ Annamalai Natarajan, for helping me learn to use Emotiv properly, for sharing information on related research, and for making a valiant effort to get the running room calendar to allow me to reserve time for my experiments.

- ◆ Nick Turk-Browne, for taking time to meet with me and share his expertise on stimulus presentation techniques.
- ◆ Matt Costa, for volunteering as a subject and giving usability feedback.
- ◆ Heather Stiles, for patiently test-driving numerous buggy iterations of the Experimenter application and for providing data and feedback.
- ◆ Ken Norman, for providing enthusiasm, assistance, time, and advice.

11 References

- Math.NET Numerics*. (2011). Retrieved 2011, from Math.NET: <http://numerics.mathdotnet.com/>
- Chang, C.-C., & Lin, C.-J. (2011). *LIBSVM - A Library for Support Vector Machines*. Retrieved 2011, from <http://www.csie.ntu.edu.tw/~cjlin/libsvm/>
- Coles, M. G., & Rugg, M. D. (1996). *Electrophysiology of Mind*. Oxford Scholarship Online Monographs.
- Dakan, R. (2010). *Review: Emotiv EPOC, tough thoughts on the new mind-reading controller*. Retrieved 2011, from Joystiq.
- Emotiv. (2009). *Extracting RAW data and other applications*. Retrieved 2010, from Emotiv Community: <http://www.emotiv.com/forum/forum4/topic401/>
- Emotiv. (2010). *Comparison with other EEG units*. Retrieved 2011, from Emotiv Community: <http://www.emotiv.com/forum/forum4/topic127/>
- Emotiv. (2010). *Developer Edition SDK*. Retrieved 2011, from Emotiv Store: <http://www.emotiv.com/store/sdk/bci/developer-edition-sdk/>
- Emotiv. (2010). *Electrode Placement*. Retrieved 2011, from Emotiv Community: <http://www.emotiv.com/forum/forum14/topic255/>
- Emotiv. (2010). *mac support?* Retrieved 2011, from Emotiv Community: <http://www.emotiv.com/forum/forum4/topic73/>
- Emotiv. (2010). *User guide by users!* Retrieved 2011, from Emotiv Community: <http://www.emotiv.com/forum/forum4/topic126/>
- Freeman, E., Freeman, E., Bates, B., & Sierra, K. (2004). *Head First Design Patterns*. O'Reilly Media.
- Freund, Y., & Schapire, R. E. (1999). Large Margin Classification Using the Perceptron Algorithm. *Machine Learning*, 277-296.
- Frijters, J. (2010). *IKVM.NET*. Retrieved 2011, from <http://www.ikvm.net/>
- Hanke, M., Halchenko, Y. O., Sederberg, P. B., Hanson, S. J., Haxby, J. V., & Pollmann, S. (2009). PyMVPA: A Python toolbox for multivariate pattern analysis of fMRI data. *Neuroinformatics*, 37-53.
- Jackson, M. (2010). *Painting Your Own Tabs - Second Edition*. Retrieved 2011, from The Code Project: <http://www.codeproject.com/KB/tabs/NewCustomTabControl.aspx>
- Johnson, M. K., Kounios, J., & Nolde, S. F. (1997). Electrophysiological brain activity and memory source monitoring. *NeuroReport*, 1317-1320.
- Liu, F. (2009). *NPatternRecognizer*. Retrieved 2011, from CodePlex: <http://npatternrecognizer.codeplex.com/releases/view/37602>
- McDuff, S. G., Frankel, H. C., & Norman, K. A. (2009). Multivoxel Pattern Analysis Reveals Increased Memory Targeting and Reduced Use of Retrieved Details during Single-Agenda Source Monitoring. *The Journal of Neuroscience*, 508-516.
- Microsoft. (2011). *IDisposable Interface*. Retrieved 2011, from MSDN: <http://msdn.microsoft.com/en-us/library/system.idisposable.aspx>
- Microsoft. (2011). *IronPython*. Retrieved 2011, from <http://www.ironpython.net/>

- Microsoft. (2011). *Timer Class*. Retrieved 2011, from MSDN: <http://msdn.microsoft.com/en-us/library/system.windows.forms.timer.aspx>
- Newman, E. (2008). *Testing a model of competition-dependent weakening through pattern classification of EEG*. Princeton University.
- Newman, E. L., & Norman, K. A. (2010). Moderate Excitation Leads to Weakening of Perceptual Representations. *Cerebral Cortex Advance Access*, 1-11.
- Novell. (2011). *Cross platform, open source .NET development framework*. Retrieved 2011, from Mono: http://www.mono-project.com/Main_Page
- Parra, L. C., Spence, C. D., Gerson, A. D., & Sajda, P. D. (2005). Recipes for the linear analysis of EEG. *NeuroImage*, 326-341.
- Root, M. (2011). *Disability and Gaming Part 3: Hardware And Where It's Going*. Retrieved 2011, from PikiGeek.
- Russel, S., & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Upper Saddle River: Pearson Education Inc.
- Schapire, R. E. (2003). The Boosting Approach to Machine Learning. *Nonlinear Estimation and Classification*, 1-23.
- Solon, O. (2011). *Mind-controlled flying rig lets you be a superhero*. Retrieved 2011, from Wired: <http://www.wired.co.uk/news/archive/2011-03/03/infinity-simulator>
- Squatriglia, C. (2011). *Thinking Your Way Through Traffic in a Brain-Control Car*. Retrieved 2011, from Wired: <http://www.wired.com/autopia/2011/03/braindriver-thought-control-car/>
- University of Waikato. (2011). *WEKA*. Retrieved 2011, from <http://www.cs.waikato.ac.nz/ml/weka/>
- Wilson. (1988). MRC Psycholinguistic Database: Machine Readable Dictionary. *Behavioural Research Methods, Instruments and Computers*, 6-11.
- Zhou, L. (2010). *The Emotiv EPOC Headset: the feasibility of a cheaper, more accessible neural imager*. Princeton: Princeton University Independent Work.

12 Appendices

12.1 Appendix A: Additional Figures

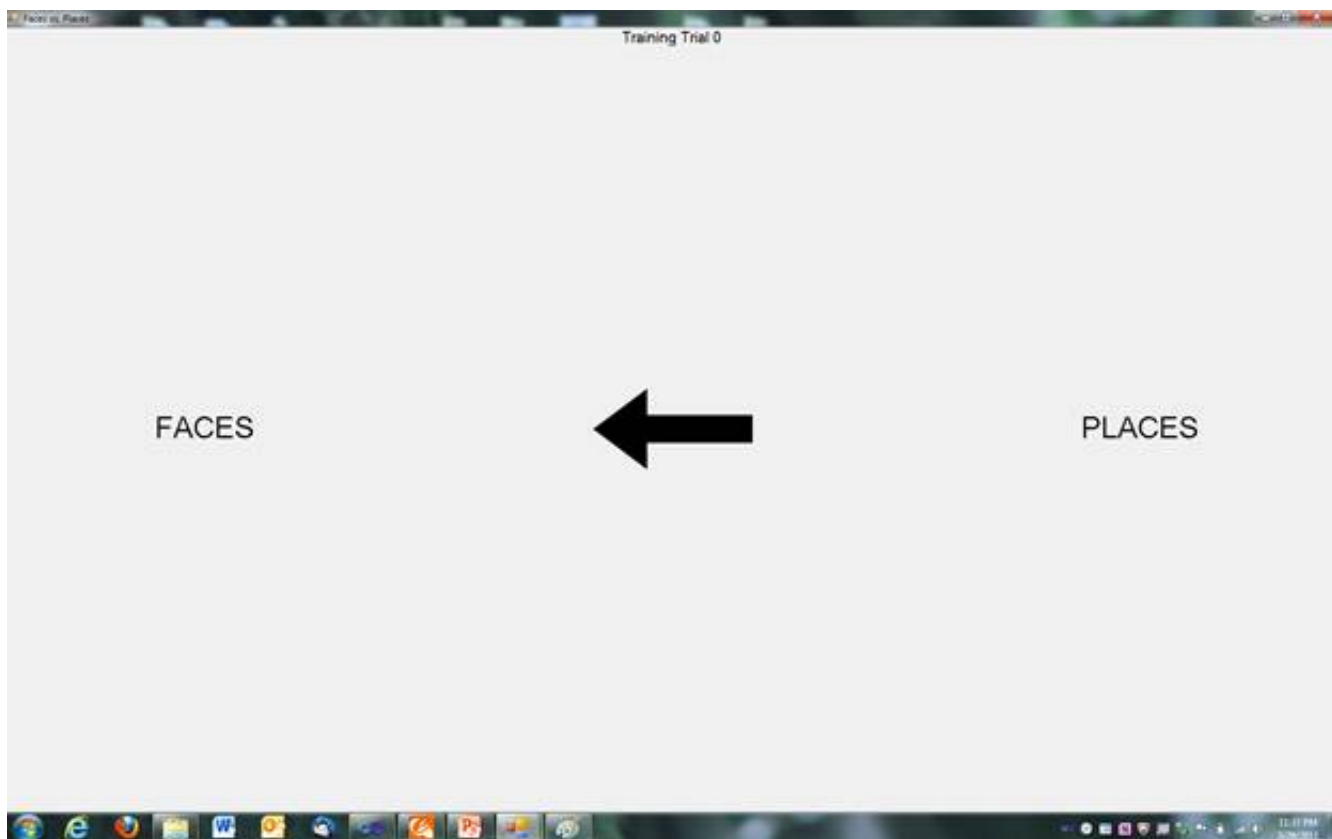


Figure 7 Training phase instruction display

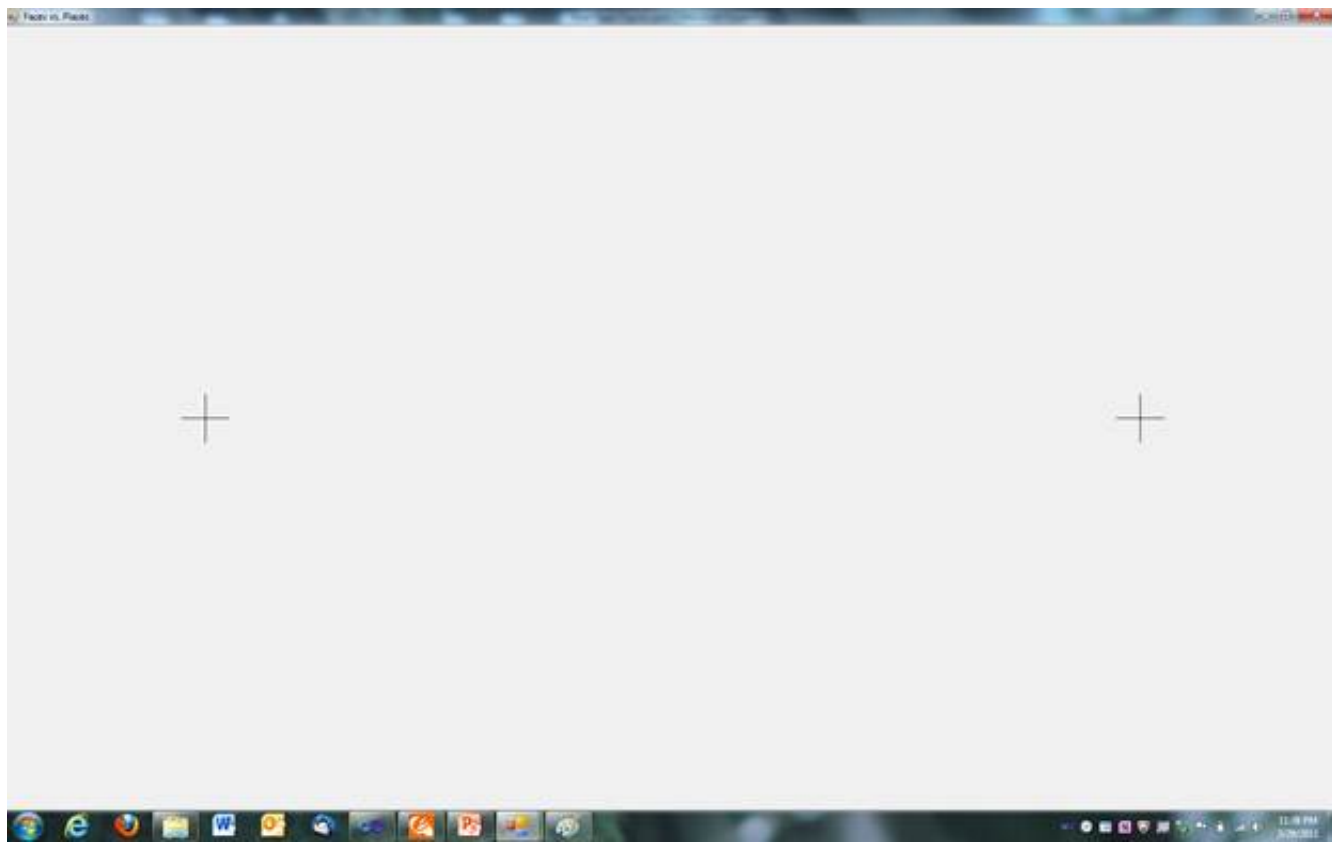


Figure 8 Side-by-side fixation cross display

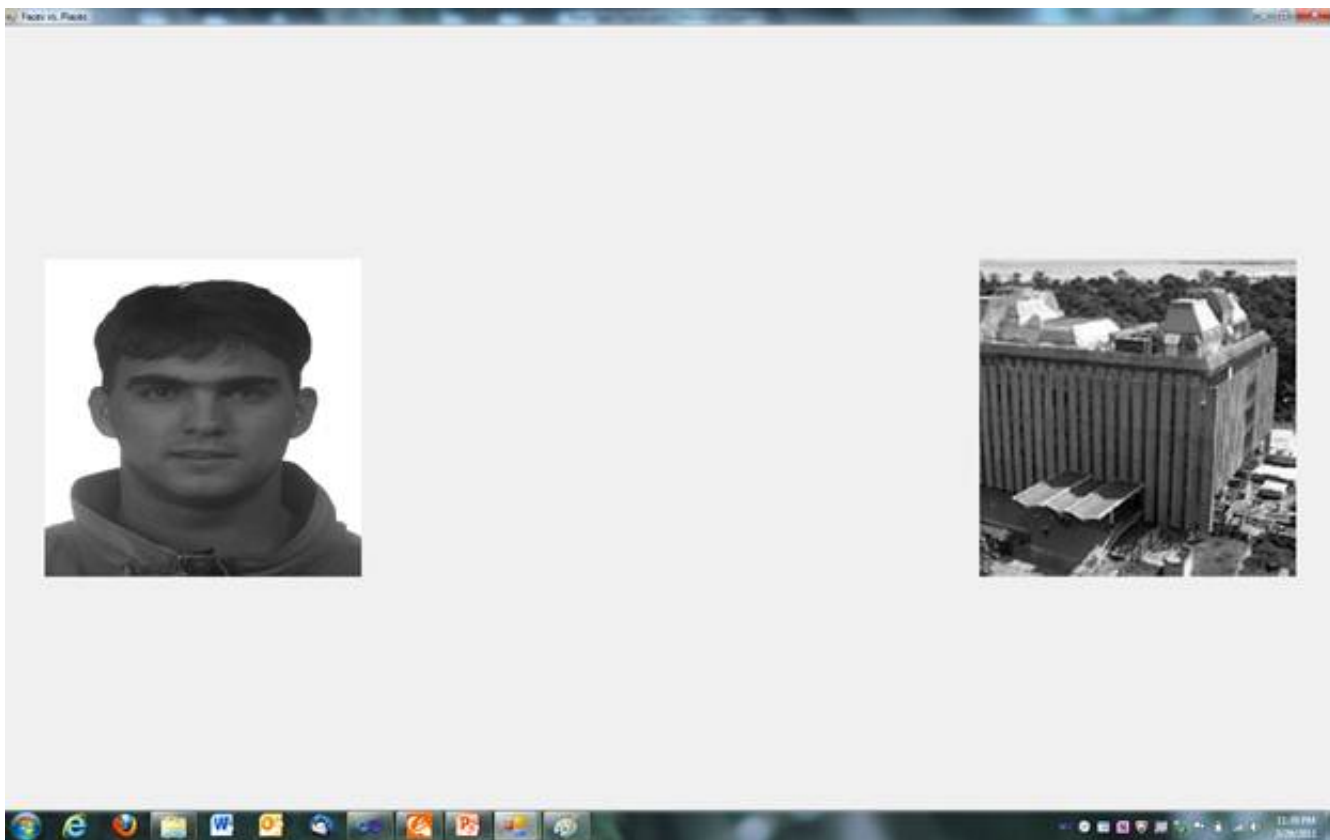


Figure 9 Side-by-side stimulus display

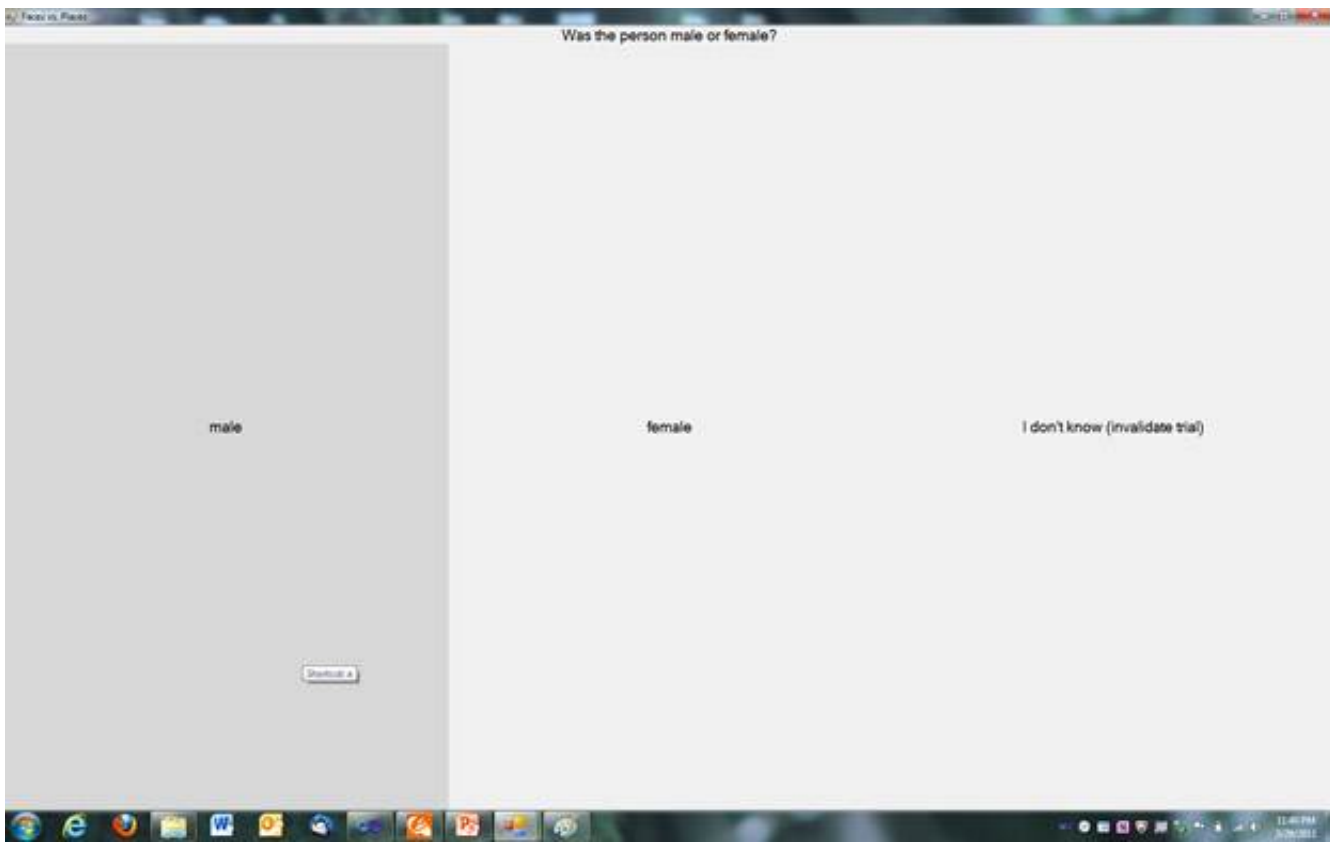


Figure 10 Choice display

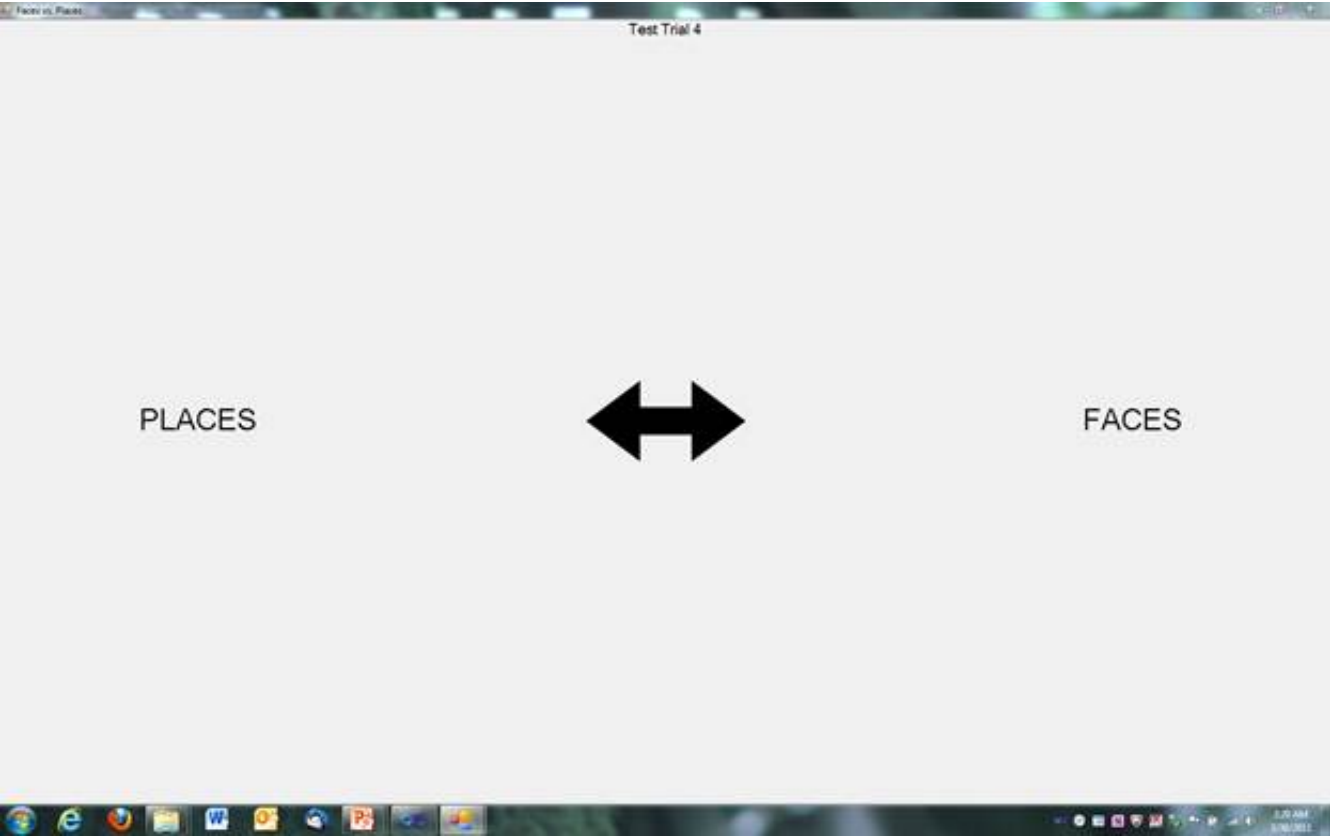


Figure 11 Test instruction display

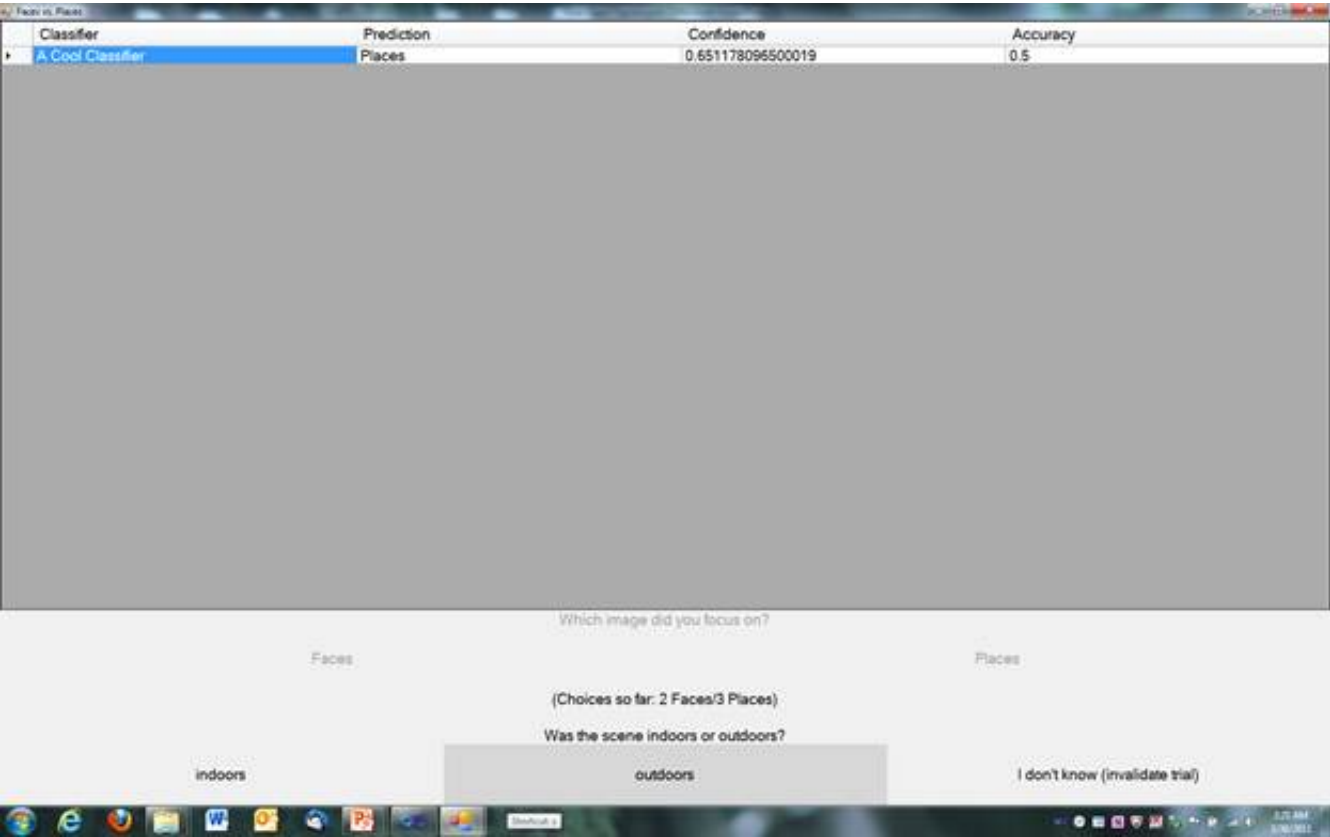
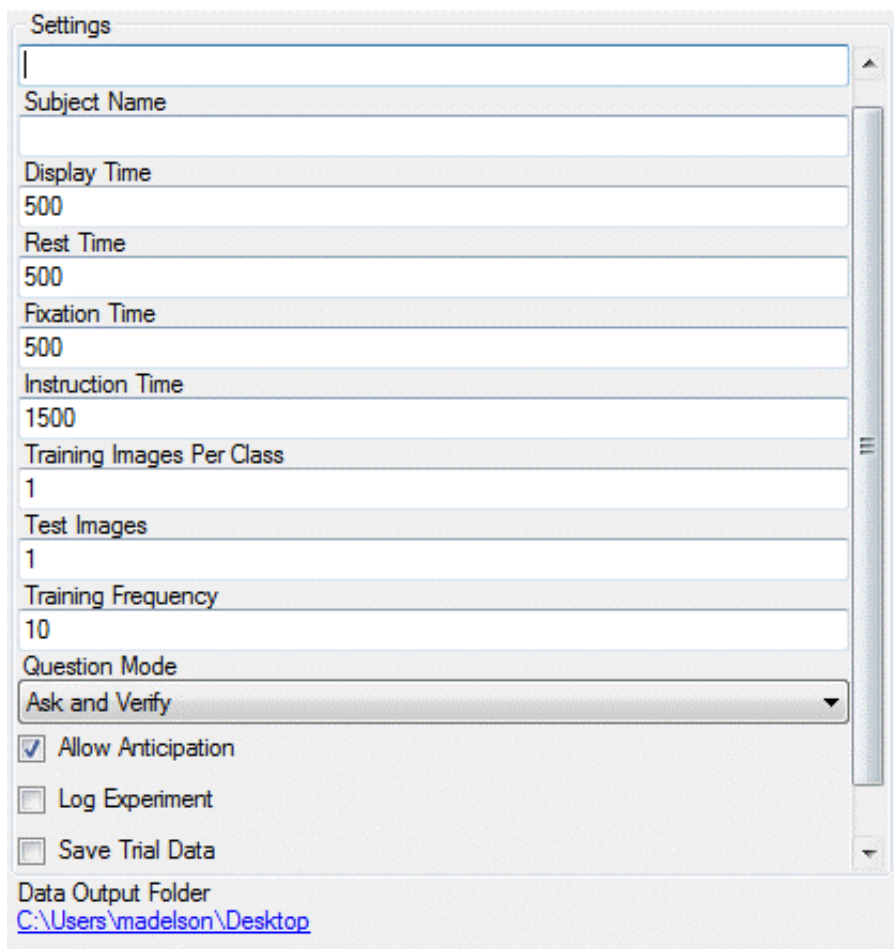


Figure 12 Test phase prediction and choice display



The screenshot shows the 'Settings' window in the Emotiv Experimenter software. It contains several input fields for configuring the experiment. The 'Subject Name' field is empty. 'Display Time', 'Rest Time', and 'Fixation Time' are all set to 500. 'Instruction Time' is set to 1500. 'Training Images Per Class' and 'Test Images' are both set to 1. 'Training Frequency' is set to 10. The 'Question Mode' is set to 'Ask and Verify' via a dropdown menu. There are three checkboxes: 'Allow Anticipation' is checked, while 'Log Experiment' and 'Save Trial Data' are unchecked. The 'Data Output Folder' is set to 'C:\Users\madelson\Desktop'.

Settings

Subject Name

Display Time

500

Rest Time

500

Fixation Time

500

Instruction Time

1500

Training Images Per Class

1

Test Images

1

Training Frequency

10

Question Mode

Ask and Verify

☒ Allow Anticipation

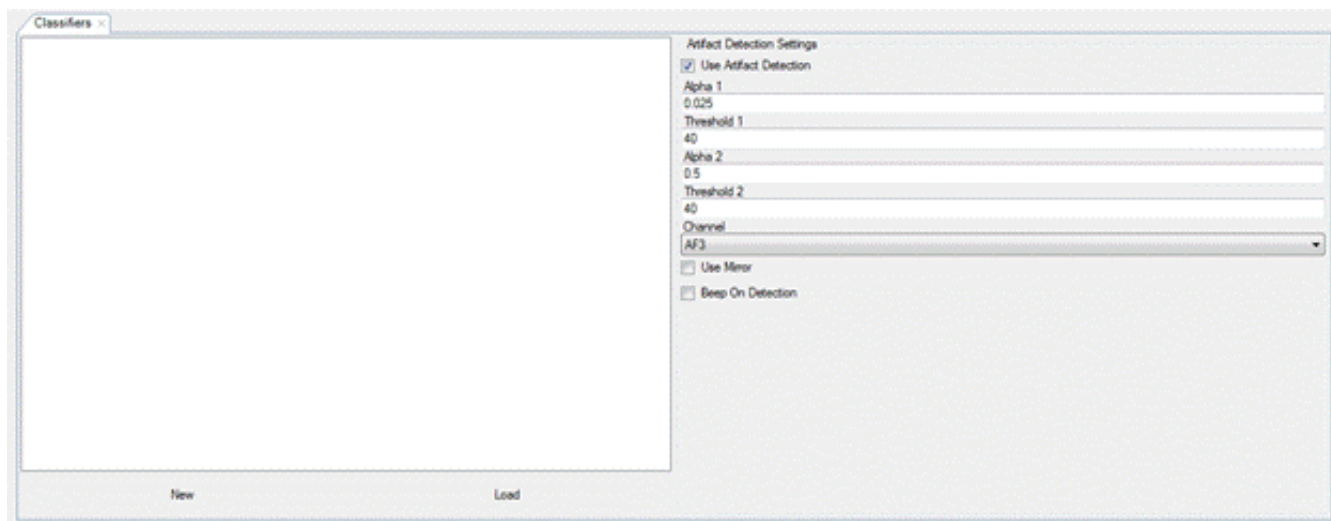
☐ Log Experiment

☐ Save Trial Data

Data Output Folder

<C:\Users\madelson\Desktop>

Figure 13 Experiment settings panel



The screenshot shows the 'Classifiers' window in the Emotiv Experimenter software. It features a large empty box on the left for classifier lists, with 'New' and 'Load' buttons at the bottom. On the right, the 'Artifact Detection Settings' are configured. 'Use Artifact Detection' is checked. 'Alpha 1' is 0.025, 'Threshold 1' is 40, 'Alpha 2' is 0.5, and 'Threshold 2' is 40. The 'Channel' is set to 'AF3' via a dropdown menu. 'Use Mirror' and 'Beep On Detection' are both unchecked.

Classifiers x

Artifact Detection Settings

☒ Use Artifact Detection

Alpha 1

0.025

Threshold 1

40

Alpha 2

0.5

Threshold 2

40

Channel

AF3

☐ Use Mirror

☐ Beep On Detection

New Load

Figure 14 Classifiers and artifact detection panel

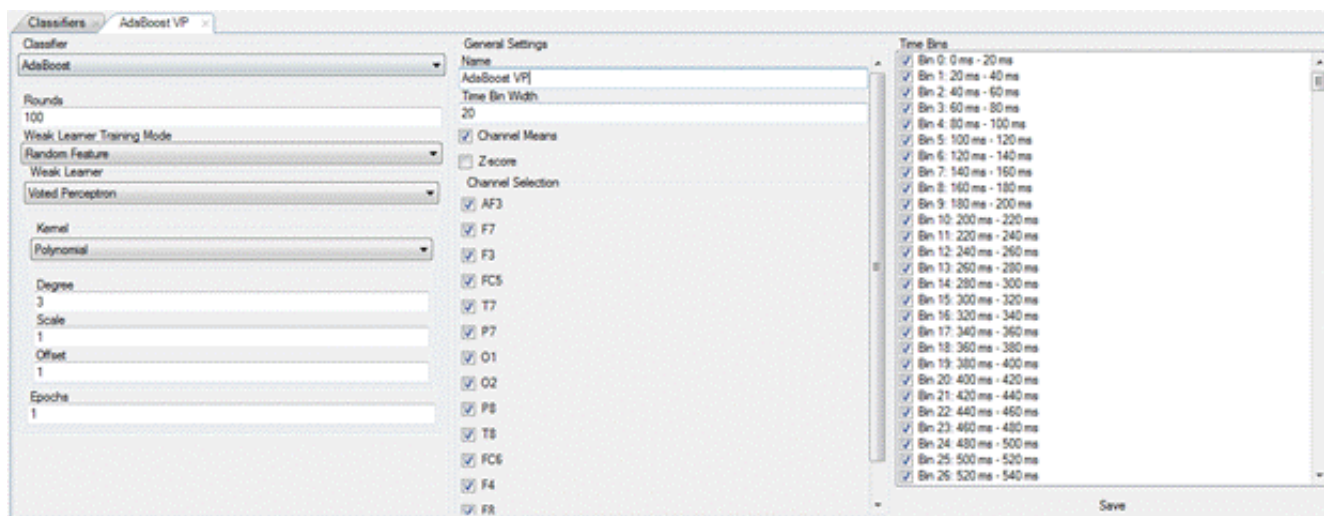


Figure 15 Individual classifier configuration tab

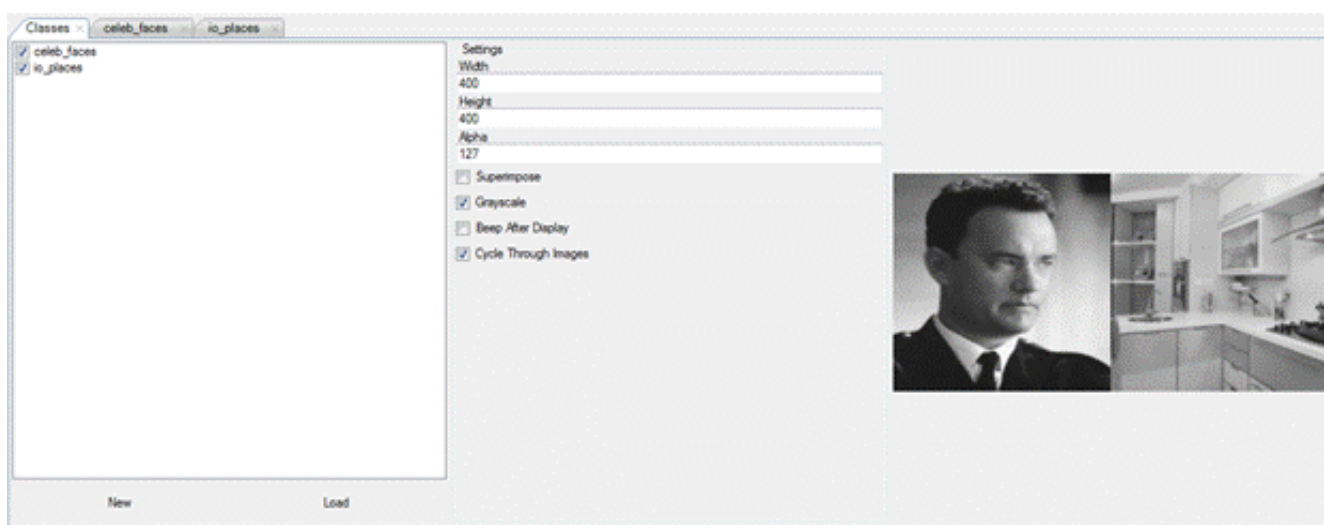


Figure 16 Classes panel

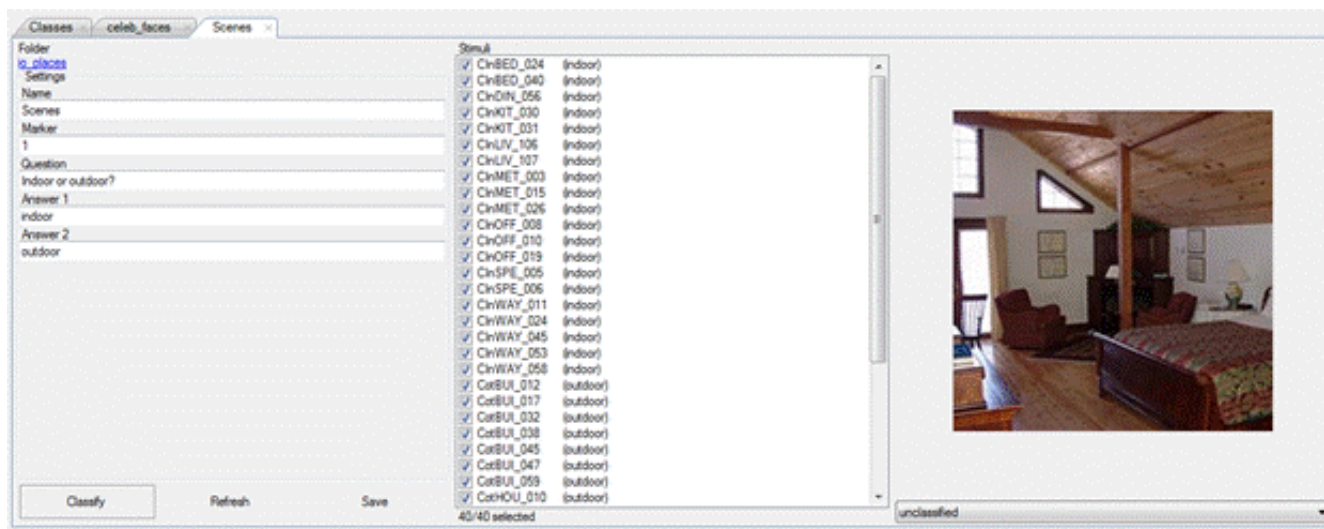


Figure 17 Individual class configuration tab

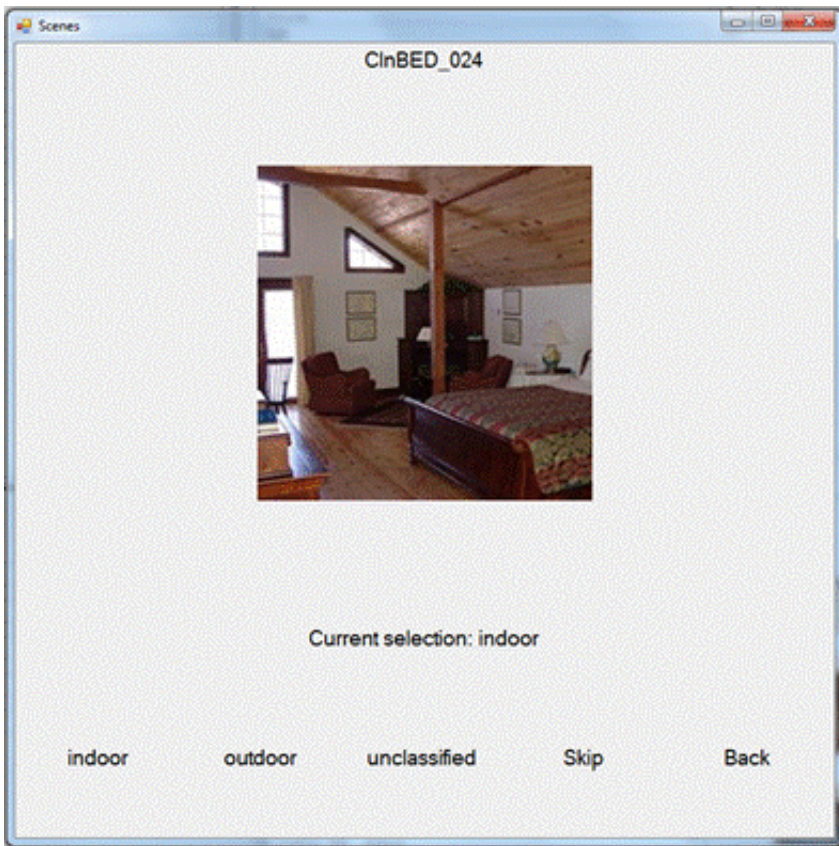


Figure 18 Image classification tool

```

IEnumerable<EEGDataEntry[]> trials = /* Get trial data from somewhere */;
int minCount = trials.Min(t => t.Length);
var examples = trials.Select(t =>
{
    var means = Channels.Values.Select(ch => t.Select(e => e[ch]).Average()).ToArray();

    return new Example(t[0].Marker, t.Take(minCount)
        .SelectMany(e => e.Select((d, i) => d - means[i]))
        .Concat(means))
    }.ToArray();

```

Figure 19 Data processing in C#

This code example demonstrates many powerful data processing features of C#. The purpose of this code is to convert a sequence (*IEnumerable*) of trials (each represented by an array of *EEGDataEntries*) into an array of *Examples* which can be fed to a classifier. This requires flattening the multi-dimensional data in each trial (some number of entries x 14 channels worth of data per entry) into a single sequence of voltage values. Along the way we also want to subtract out the mean voltage value along each channel, and then append these mean values to the end of the feature sequence. Furthermore, since by chance some trials have a few more examples than others, we need to ensure that the resultant example feature sets are trimmed such that the features in each example line up regardless of the length of the original trial. In many popular programming languages, this would require many lines of code with multiple loops. In C#, we require neither.

This code may seem arcane at first, but it is easy to break down. The code is peppered with inline anonymous functions (also called lambda functions). These functions are designated with the `=>` operator, which can be interpreted as `goes to`. Thus the anonymous function `t => t.Length` takes one argument

(*t*) and returns the *Length* property of *t*. In this case, *t* is an array of *EEGDataEntries*, but although C# is statically typed, there is no need to declare the type of *t* since the compiler can infer it from context. Similarly, note that the *examples* and *means* variables are declared with type *var*. This is a shortcut for declaring the types of these variables, which in this case are *Example[]* and *double[]* respectively. The compiler permits the use of *var* since those types can be inferred from context.

The next step in unpacking this code is to note the variety of functions in C# which operate on (and sometimes return) sequences (*IEnumerables*). These functions often take another function as an argument. For example, the *Select* function for a sequence of items of type *T* takes a function *selector* whose argument is of type *T*. *Select* then iterates over the sequence, evaluating *selector* for each item and returning a new sequence of the results.

Consider the code for determining the mean value along each EEG channel:

```
var means = Channels.Values.Select(ch => t.Select(e => e[ch]).Average()).ToArray();
```

Channels.Values is an array of all channel values. We call *Select* on this array, passing in the anonymous function:

```
ch => t.Select(e => e[ch]).Average()
```

as the *selector* function. *Channels.Values* is a sequence of items of type *Channel*. Thus, the *selector* function takes a *Channel* (*ch*) as an argument. The *selector* function then calls *Select* again, this time on *t*, which is an array of *EEGDataEntries*. The *selector* function for this call to *Select*:

```
e => e[ch]
```

thus takes an *EEGDataEntry* (*e*) as its argument. This function returns the voltage value of *e* at channel *ch*. Thus, the *Select* call returns a sequence of real-value voltages. We then call the *Average* function on this sequence to compute its mean. Thus, for each channel in *Channels.Values*, we have selected the mean voltage value of the *t* along that channel. Finally, a call to the *ToArray* function converts this sequence of mean values to an array, allowing for random access. This step is also important because the sequences returned by calls to *Select* and the like are lazily evaluated. That is, they postpone computing the each item in the sequence until that item is explicitly requested. This can be very efficient, because it avoids allocating space for and computing entire sequences when only the first few items are used or when the sequence is immediately filtered by another call to a function like *Select*. On the other hand, this is inefficient for a sequence which will be iterated over more than once, since the computation is repeated on each iteration. Calling *ToArray* avoids this by copying the final sequence into a non-lazy data structure, thus allocating memory but computing each sequence element only once. At first glance, this sort of code might seem overly complex to develop. However, the Visual Studio IDE's Intellisense and autocomplete features actually make this style of coding faster and more intuitive than a traditional imperative style.

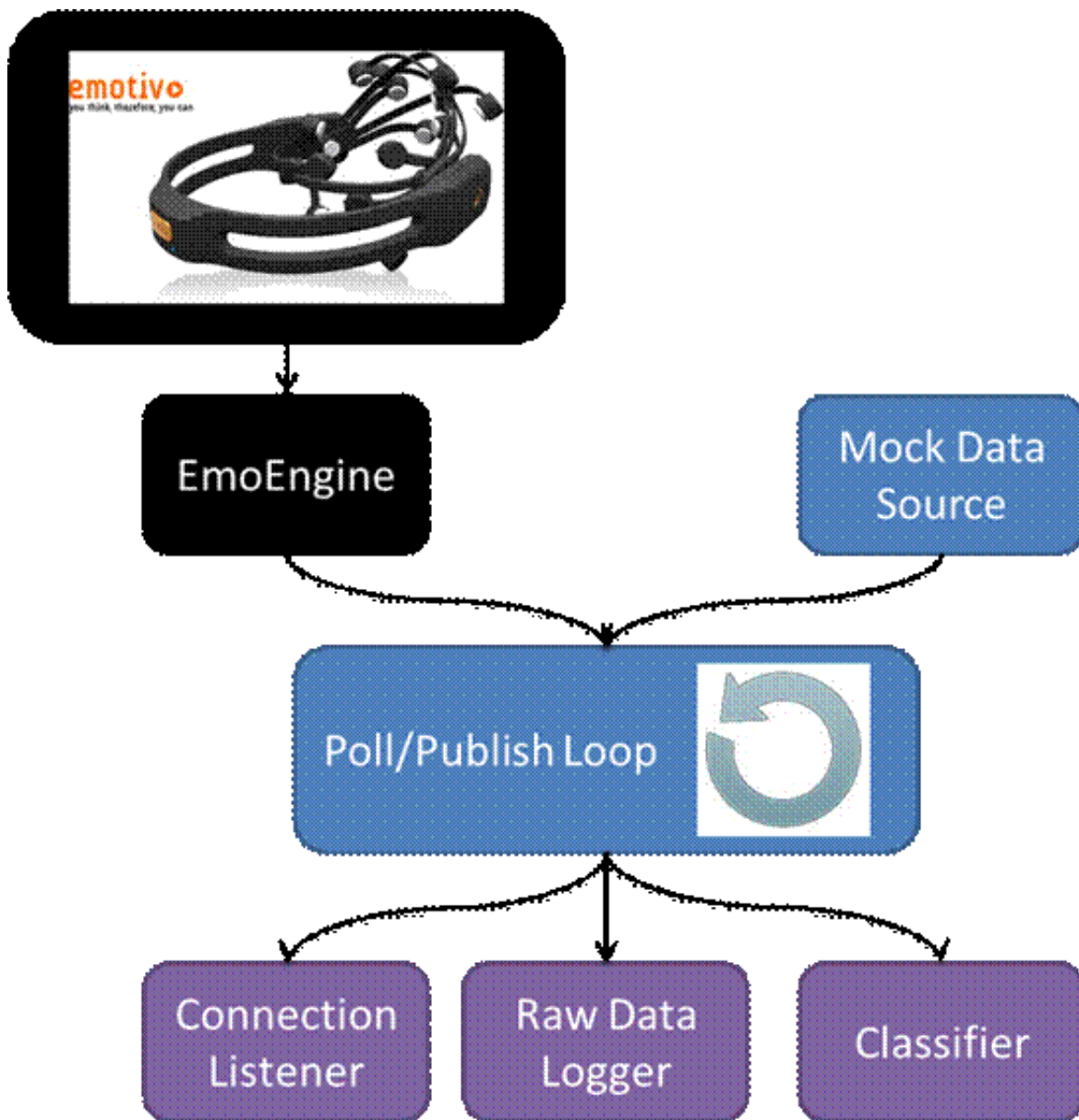


Figure 20 The publish/subscribe interface to the Emotiv data stream

This diagram shows the structure of Experimenter's interface to the Emotiv headset. The Emotiv-provided EmoEngine (or alternatively a mock data source) acts as a source of raw EEG data which is queried periodically by the poll/publish loop. This data is then converted to a more usable form and published to all subscribers. If the poll/publish loop detects a new connection or a connection failure, this information is also published. Three possible types of subscribers are shown.

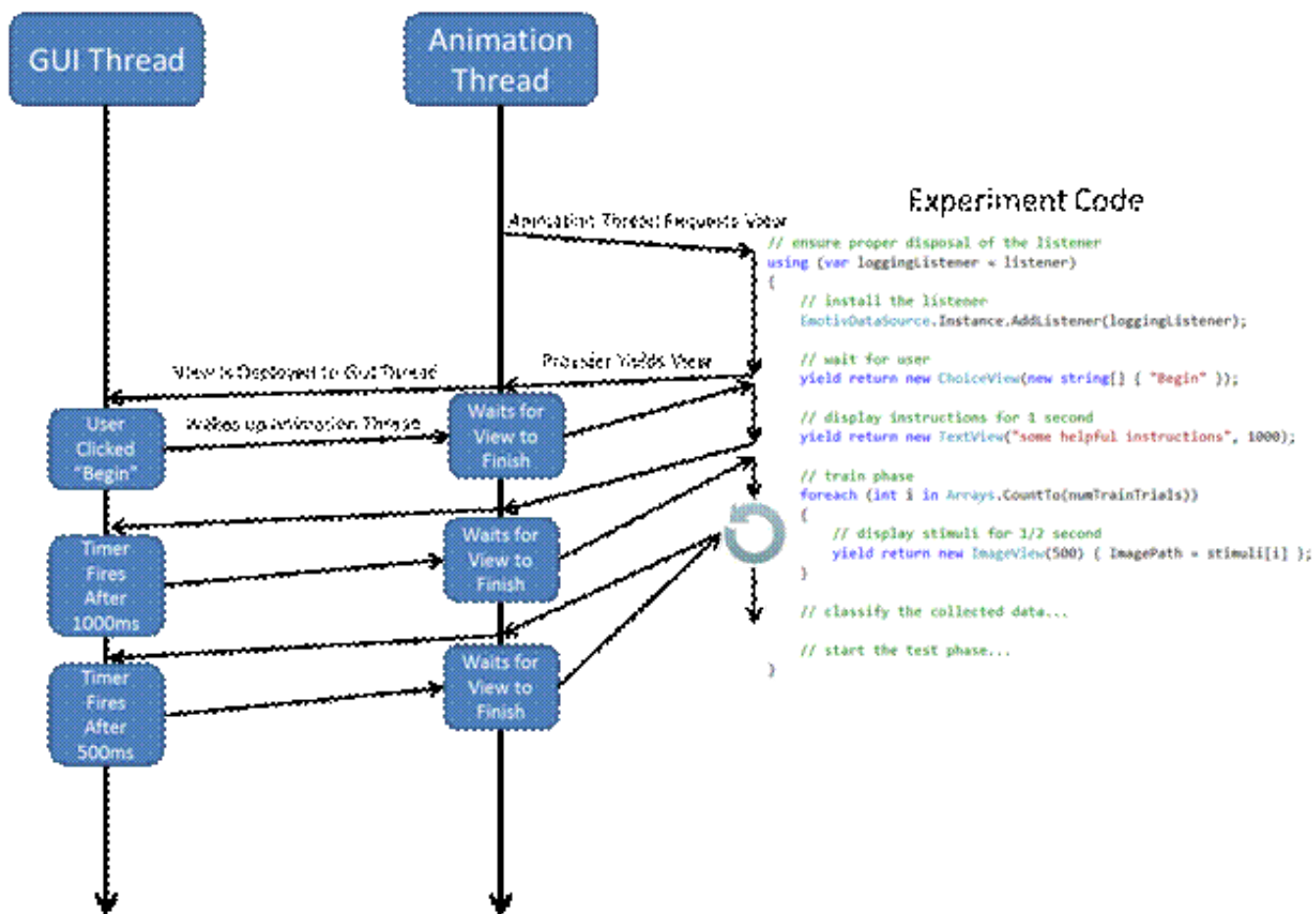


Figure 21 The yield-view presentation system

This diagram shows how the yield/view presentation system maps clean, sequential experiment code onto the event-based GUI framework. The animation thread repeatedly requests views and deploys them to the GUI thread to be displayed. The animation thread then blocks until some event in the GUI thread causes the view to finish, at which point the process repeats. Note that the experiment code is free to use any sort of control flow structures (a loop is shown here). Also, the use of the *using* statement to declare the *loggingListener* guarantees that the listener will be properly disposed of (unsubscribed) even if the experiment terminates prematurely.


```
[Description("Image display configuration", DisplayName = "Settings")]
class ImageDisplaySettings
{
    [Parameter("The maximum width (in pixels) of the displayed image during an experiment", DisplayName = "Width", DefaultValue = 400, MinValue = 1)]
    public int ImageWidth { get; set; }

    [Parameter("The maximum height (in pixels) of the displayed image during an experiment", DisplayName = "Height", DefaultValue = 400, MinValue = 1)]
    public int ImageHeight { get; set; }

    [Parameter("The transparency value [0 - 255] of the overlaid image", DefaultValue = 127, MinValue = 0, MaxValue = 255)]
    public int Alpha { get; set; }

    [Parameter("Should the images be superimposed or displayed side by side?", DisplayName = "Superimpose", DefaultValue = false)]
    public bool SuperimposeImages { get; set; }

    [Parameter("Should the images be displayed in black and white?", DisplayName = "Grayscale", DefaultValue = false)]
    public bool UseGrayscale { get; set; }

    [Parameter("Should a beep be played at the end of each trial?", DisplayName = "Beep After Display", DefaultValue = false)]
    public bool Beep { get; set; }
}
```

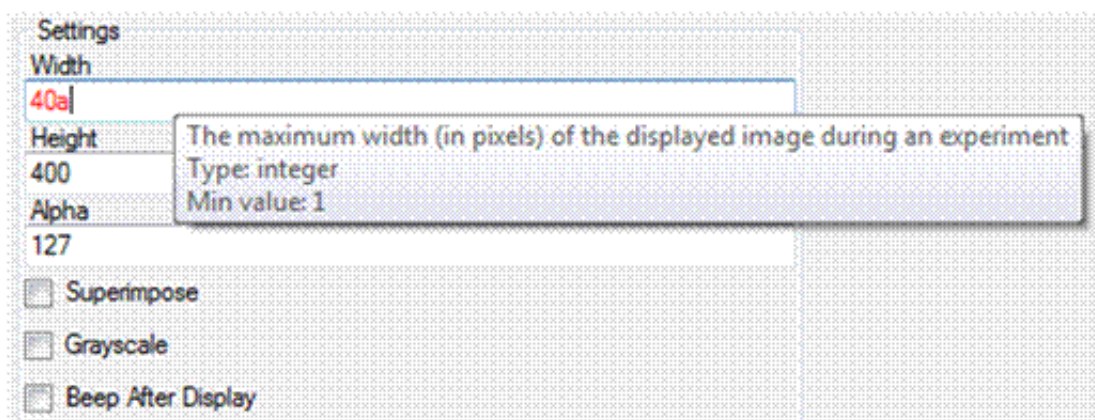


Figure 22 The parameter-description system

This figure demonstrates how clean object code decorated with the custom *Parameter* attribute can be used to dynamically construct a graphical component for configuring the object. Note that each control in the dynamically constructed component displays a helpful tooltip when moused over. The red text in the *Width* box demonstrates the component's validation features: *Width* only accepts an integer, so the input 40a is invalid. This validation procedure also checks inputs against the stated *MinValue* and *MaxValue*, if applicable.

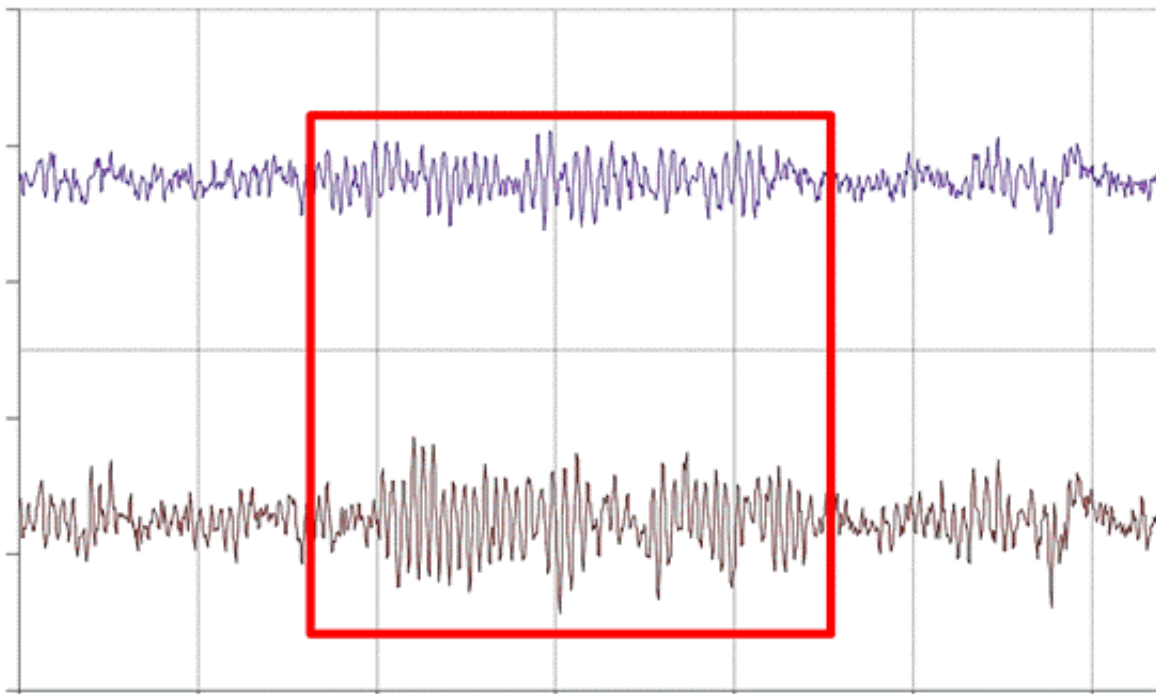


Figure 23 Eyes closed response on the O1 and O2 channels

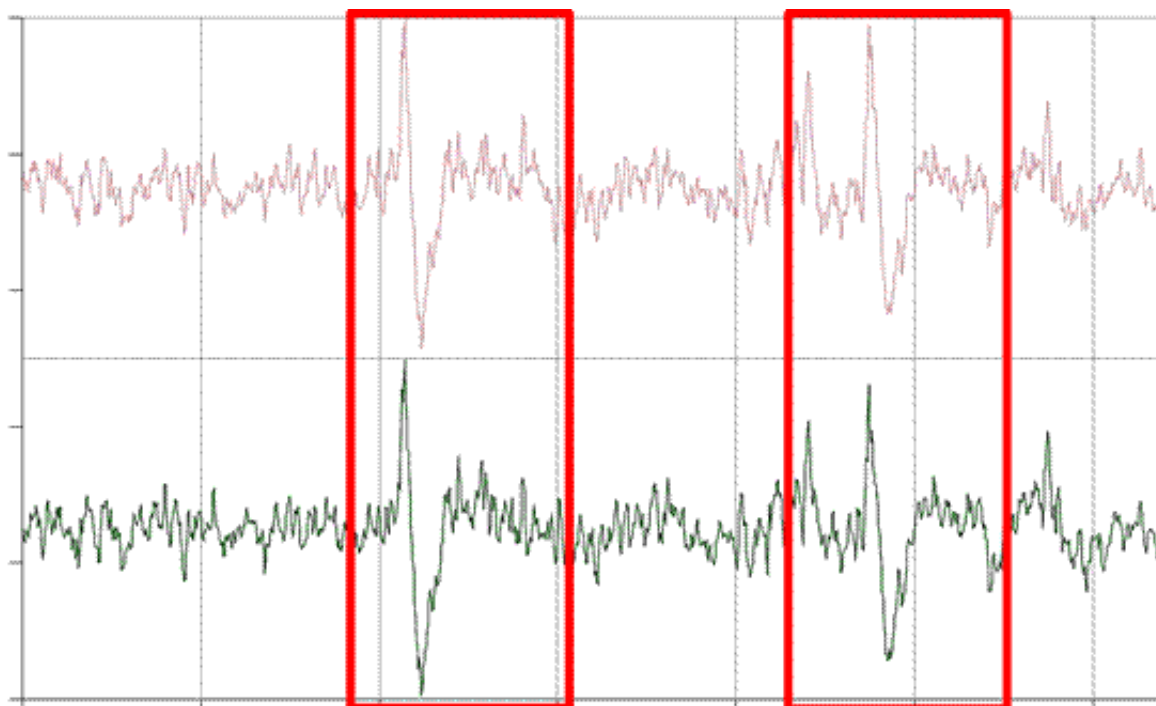


Figure 24 Eye blink response on the AF3 and AF4 channels

```

??? public static void main(String[] args)
??? {
???     ??? int num1 = 14, num2 = 16, numProbs = 1000;
???     ??? Random r = new Random();
???     ??? Set<String> set = new HashSet<String>();
???     ??? for (int i = 0; i < numProbs; i++)
???     ??? {
???         ??? int target = (i % 2) == 0 ? num1 : num2;
???         ??? int a, b, c, sign;

```

```

String prob;
do
{
    a = r.nextInt(num2) + 1;
    if (a == num1 || a == num2)
        a++;
    b = r.nextInt(num2) + 1;
    sign = r.nextBoolean() ? 1 : -1;
    if (sign > 0 &&
        (b == num1 || b == num2))
        b++;
    c = target - a - sign*b;
    prob = String.format("%d %c %d %c %d",
        a,
        (sign > 0 ? '+' : '-'),
        b,
        (c > 0 ? '+' : '-'),
        Math.abs(c));
} while (c == 0
        || c == num1
        || c == num2
        || set.contains(prob));
set.add(prob);
System.out.println(prob);
}
}

```

Figure 25 Expression generation code

A simple Java program for randomly generating 1000 expressions with values 14 and 16. A set is used to avoid generating duplicate expressions. Also, no terms are allowed to be the same as either target value to avoid too-easy expressions like $14 + 5 = 5$. Similarly, no terms are allowed to be zero.

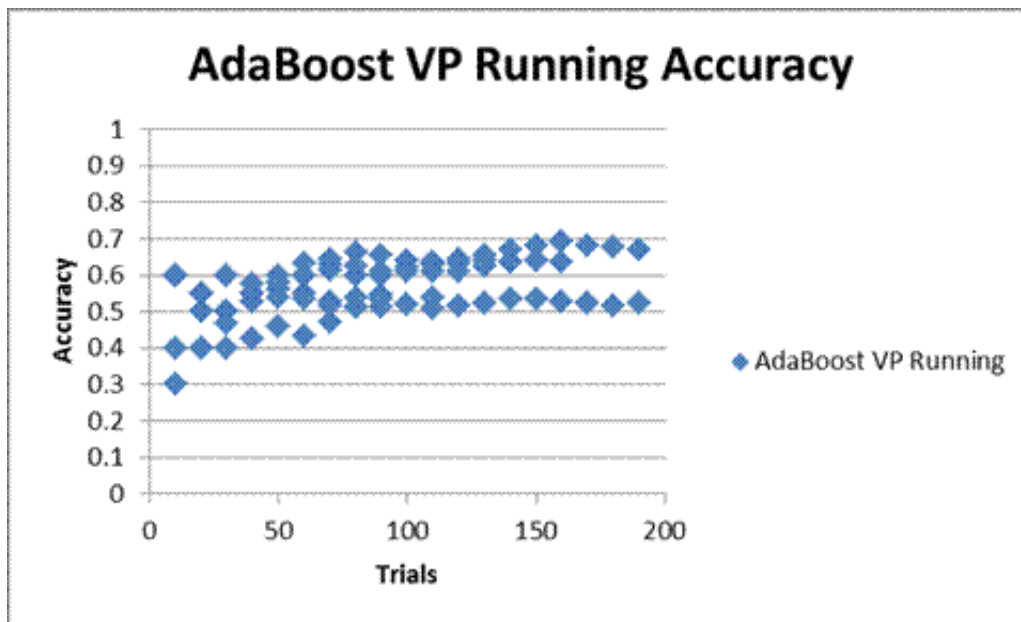


Figure 26 Version 3 AdaBoost VP online accuracy

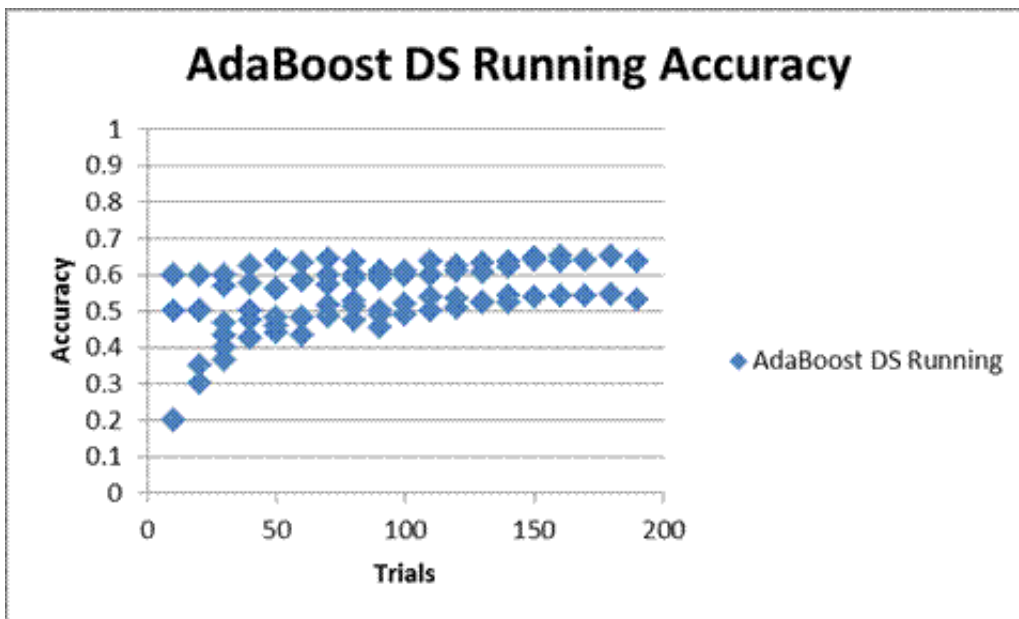


Figure 27 Version 3 AdaBoost DS online accuracy

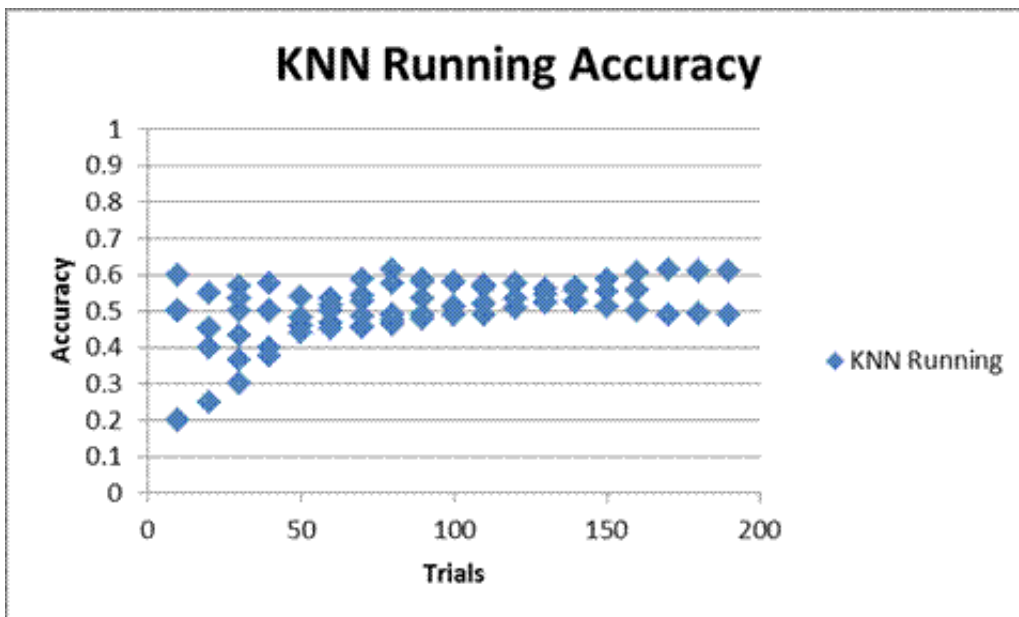
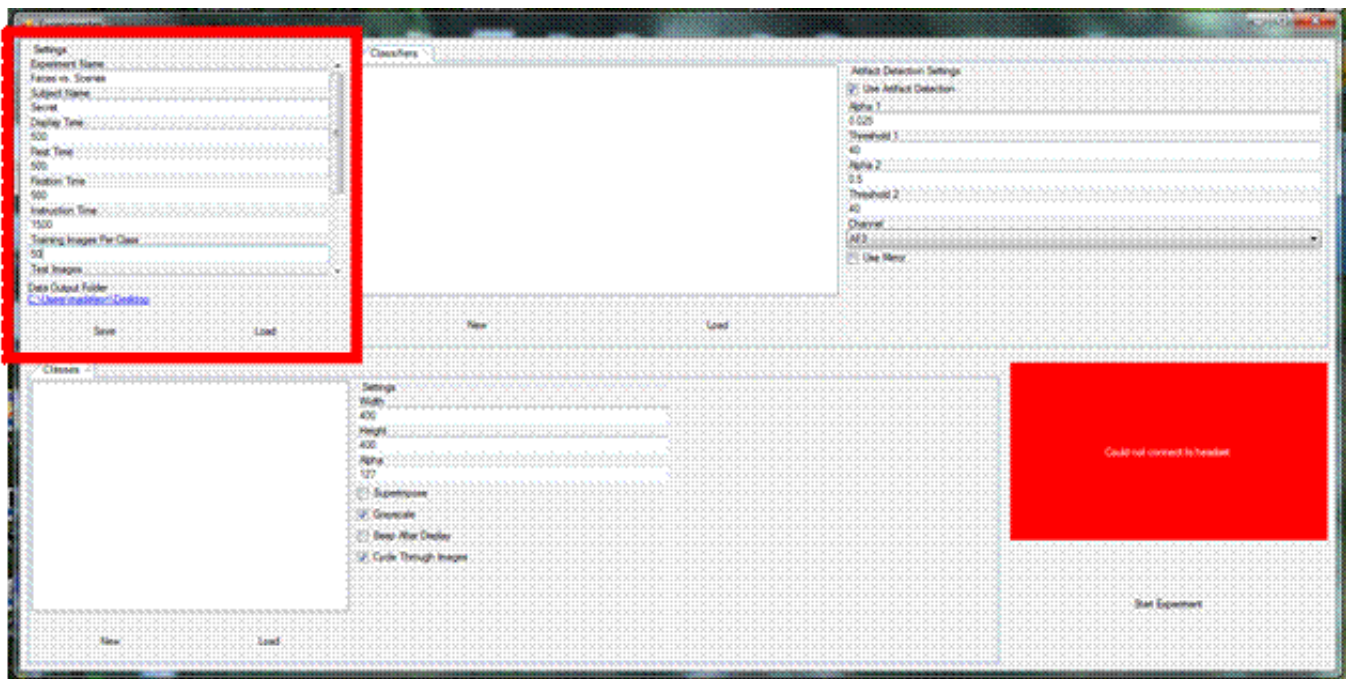


Figure 28 Version 3 KNN online accuracy

12.2 Appendix B: Walkthrough

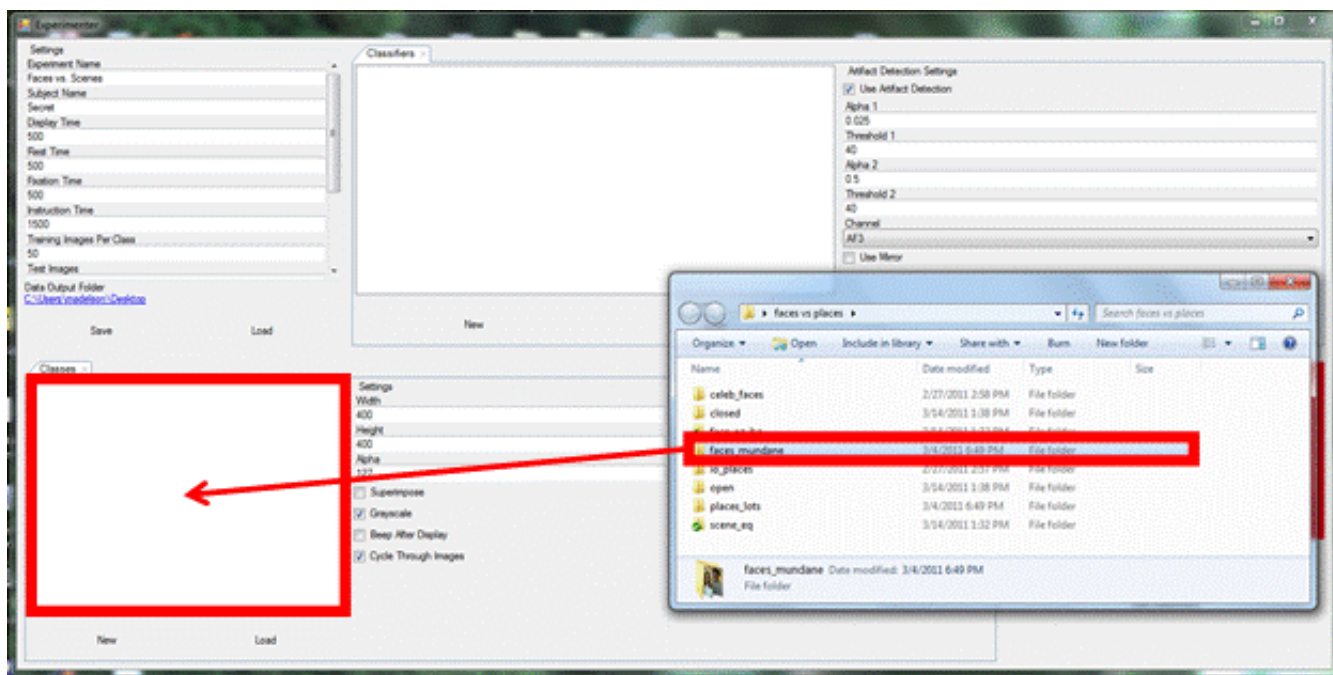
This section illustrates a walkthrough of using Experimenter to configure and run a faces vs. scenes experiment.

1. Experiment Configuration



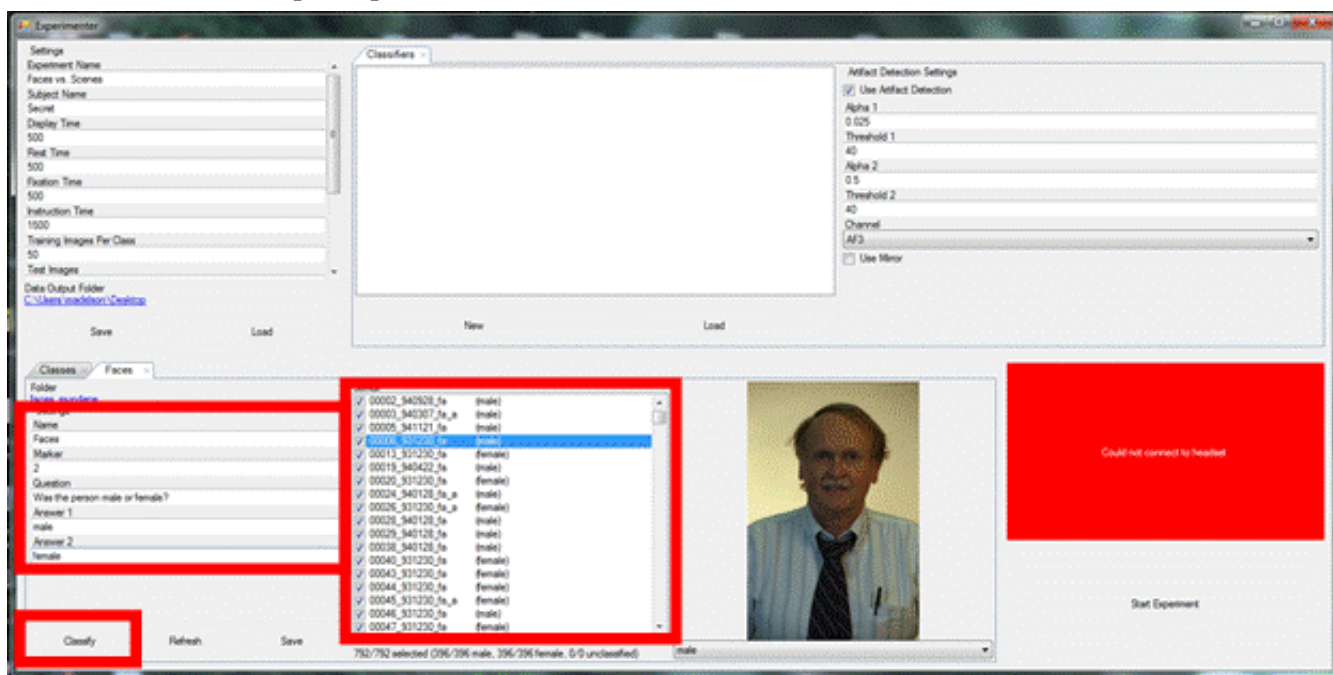
The experiment settings region allows for the configuration of a number of experimental parameters, including timing information and the number of stimuli to be displayed during each phase of the experiment. Also important is the ability to specify the types of output to be produced. By default, no output is saved since extraneous output is irritating when trying to test out an experiment. For real experiments, though, the *Log Experiment* option is highly recommended. There are also two data output formats. *Save Raw Data* logs *all* data collected during the experiment. However, this means that test phase data is marked with the unknown marker (-2) rather than with the correct class marker. This is because the correct class of the test data is not known by the application until the user declares it following prediction. In most cases, the *Save Trial Data* option is more useful. When this option is turned on, only correctly marked data collected during valid trials is logged. Both save modes can be run in parallel, generating two output files. Since each data point contains timing information, it is easy to match up saved trials with the raw data time series, which can be useful for some kinds of offline analysis. All output is saved to the folder indicated under *Data Output Folder*. Clicking on this link allows this folder to be changed. Finally, the *Save* and *Load* buttons allow experiment settings to be saved between uses of the application. Note that, depending on the size of the application on the screen, it may be necessary to scroll to see all of the configuration options available.

2. Load Stimuli



The easiest way to load a class of stimuli into the application is to drag and drop a folder containing the stimuli into the box in the *Classes* section of the form (this can also be accomplished via the *New* and *Load* buttons). All image files in the folder (.bmp, .jpg, .jpeg, .png, and .tiff files) are treated as stimuli. Text (.txt) files are treated as containing text stimuli. In this case, every non-empty line of text is treated as a separate stimulus. If the settings for the stimulus class have already been configured and saved to the folder, they are loaded as well.

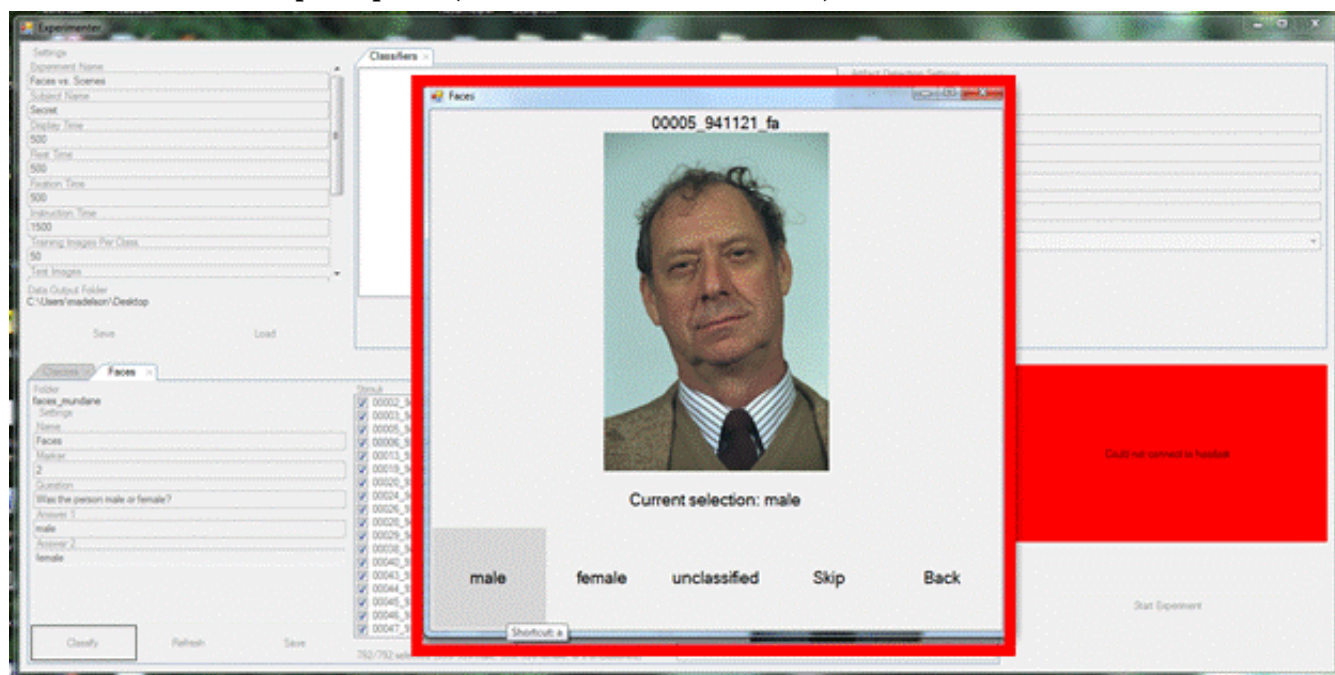
3. Stimulus Class Setup: Step 1



When a stimulus class is loaded, a new tab is created for the class. The controls on this class allow various class parameters to be configured. The *Name* field is used to refer to the class during experiments. The *Marker* field acts as a numerical identifier for the class and is used to tag EEG data while a stimulus from the class is being displayed. The *Question* and *Answer* fields are used to specify the subcategories for stimuli in the class. For example, a faces class might have ♡Was the person male or female?♡, ♡male♡, and ♡female♡ in these fields. Clicking on a stimulus in the

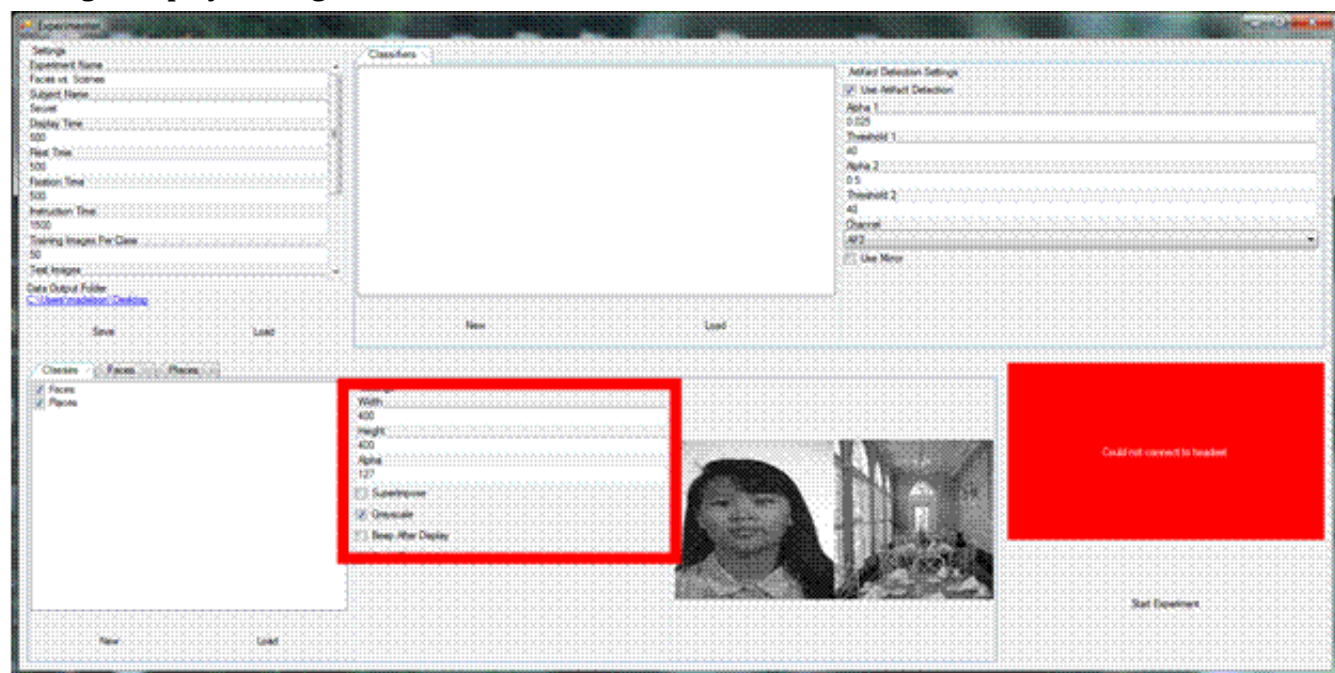
list of stimuli causes that stimulus to be displayed on the right side of the tab. The drop down list beneath the stimulus can then be used to set the stimulus's subcategory. Unchecking the checkbox next to a stimulus excludes that stimulus from being used in experiments.

4. Stimulus Class Setup: Step 2 (Newman & Norman, 2010)



Since specifying the subcategory of hundreds of images is rather tedious, Experimenter provides a tool to assist with this process which can be accessed via the *Classify* button on the class's tab. The tool allows the user to run through the images, classifying them one by one in rapid succession. Keyboard shortcuts to the various buttons make this process very efficient.

5. Image Display Configuration

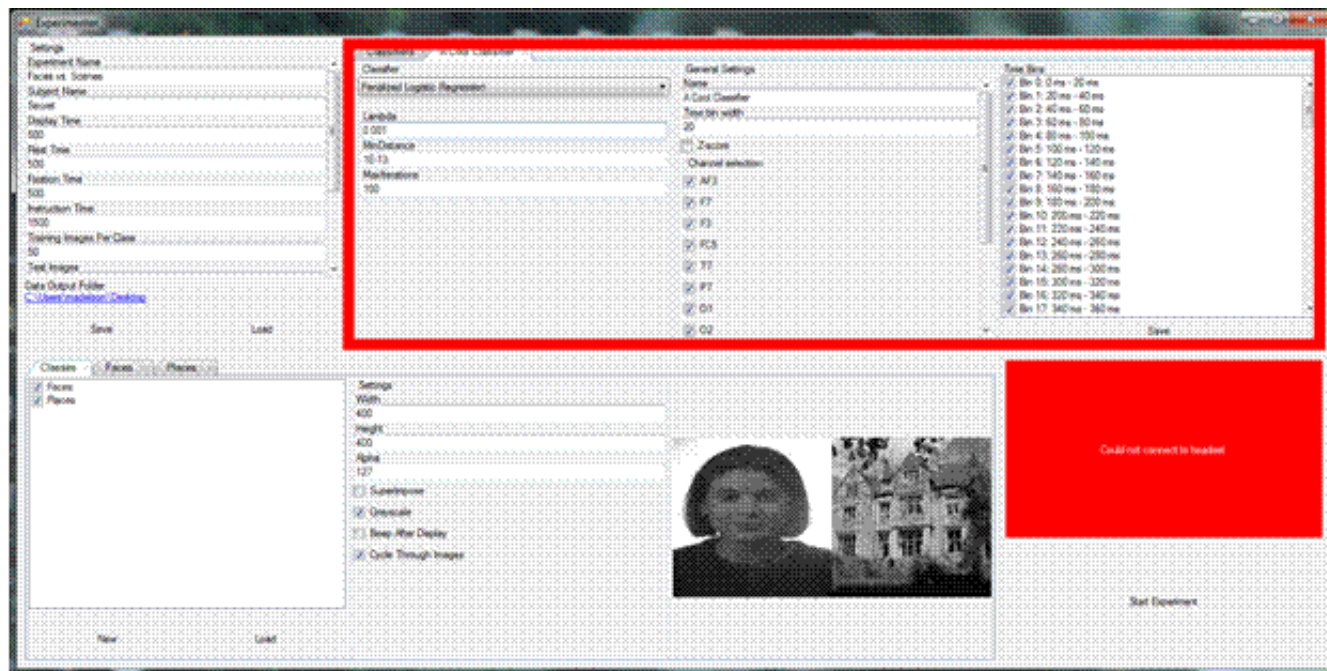


After the stimuli are loaded, the next step is to configure how they are displayed. The image display configuration panel contains several controls for this purpose. If the *Cycle Through Images* box is

http://compmem.princeton.edu/experimenter/ExperimenterReport.html#_Toc290642081

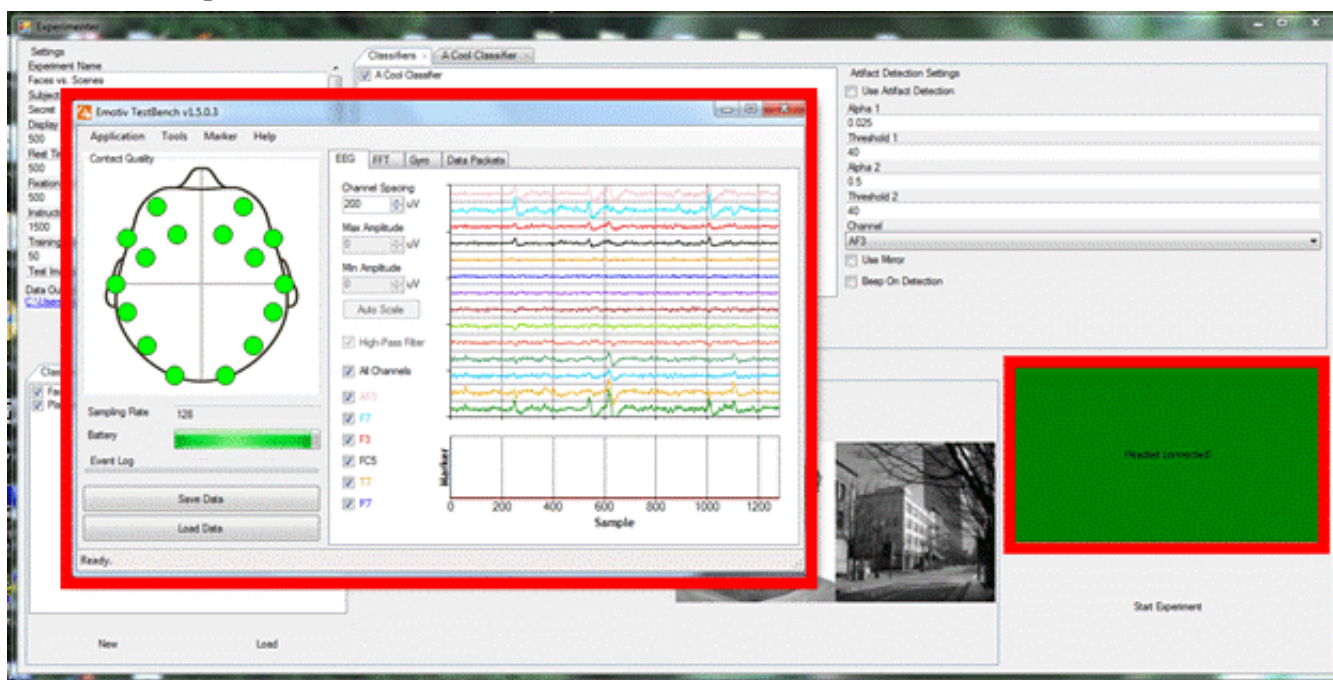
checked, the sample images displayed on the right change periodically, which can be helpful to determine whether the settings are appropriate.

6. Classifier Setup



Classifiers can be created/loaded via the *New* and *Load* buttons on the *Classifiers* tab. Like each stimulus class, each classifier has its own tab where it can be configured. This tab contains settings particular to the classifier as well as general feature selection options.

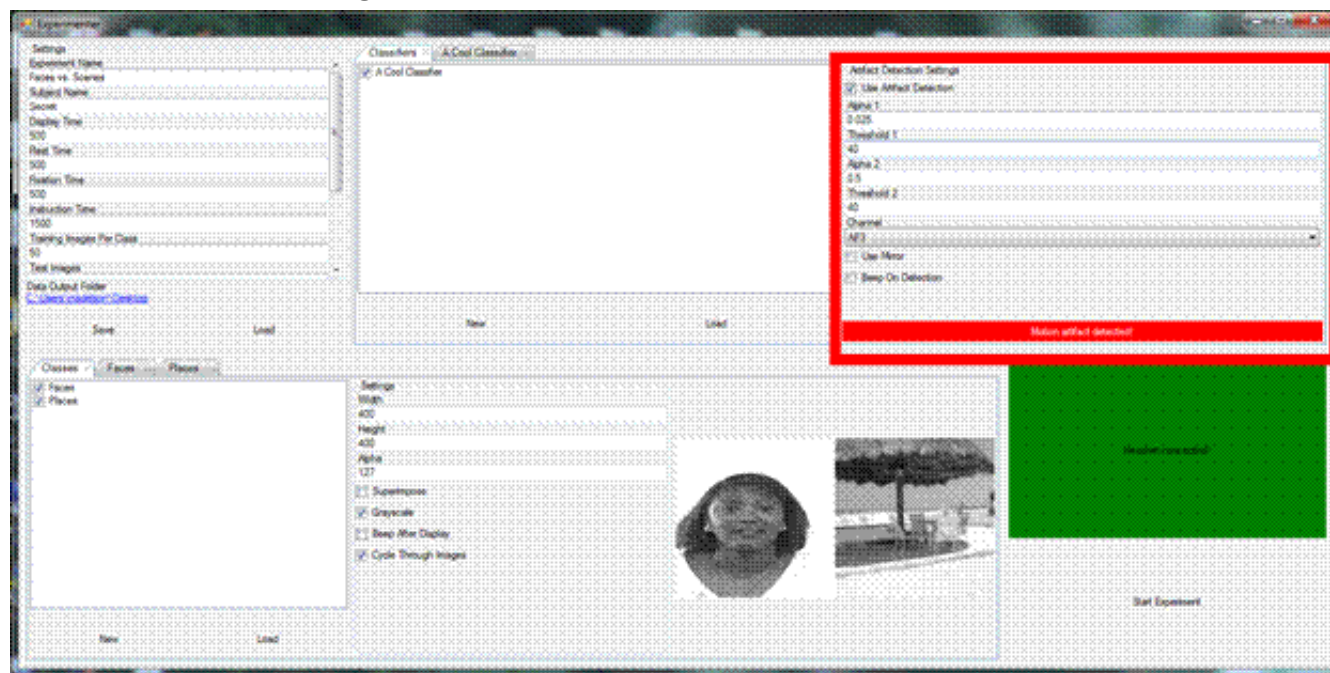
7. Headset Setup



When the Emotiv headset is connected, the red panel on the right of the form turns green to register the connection. It is recommended that the free Emotiv TestBench application be used to get the headset properly situated on the subject. The experiment should not begin until all circles in the

display have turned green.

8. Artifact Detection Configuration



With the headset plugged in, artifact detection can be configured. The artifact detection panel on the *Classifiers* tab allows various parameters of the detection algorithm to be edited. The box at the bottom of the panel turns red when an artifact is detected. This feedback can be used to tune the algorithm.

When all of these steps are complete, the experiment can be started by hitting the *Start Experiment* button.

12.3 Appendix C: Experimenter Documentation

The main body of this report describes how Experimenter's code base approaches several high level design issues. Meanwhile, auto-generated Doxygen documentation (accessible at the time of this writing from <http://www.princeton.edu/~madelson/experimenter/>) provides a resource for browsing the code at a file-by-file level. Thus, this section intends to provide project and namespace-level documentation.

The Experimenter code base is comprised of three projects: *Emotiv*, *GUI*, and *Testing* which are contained within a single Microsoft Visual Studio solution. Each project compiles to a separate assembly: *Emotiv* compiles to a dynamic link library (DLL), while *GUI* and *Testing* compile to executable files.

The *Emotiv* project provides core library functionality for applications working with the Emotiv headset. All project files reside in one of four subfolders, each of which typically corresponds to a separate

namespace.

- ❖ The *Common* subfolder contains generally useful utility classes. Since these classes are used so commonly, they are located in standard *System* namespaces rather than in an *MCAEmotiv* namespace. Of particular note are the *ParameterAttribute* and *DescriptionAttribute* classes, which implement the parameter/description system.
- ❖ The *Interop* subfolder contains classes in the *MCAEmotiv.Interop* namespace. These classes are related to interacting directly with the Emotiv headset. The most important classes are the *IEEGDataSource* and *IEEGDataListener* interfaces, which describe the publish/subscribe functionality that provides access to the Emotiv data stream. A data source for the headset can be accessed via the static *EmotivDataSource.Instance* property, while the *MockEEGDataSource* class acts as a mock that can be used for testing. Finally, data is delivered to listeners in the form of instances of *EEGDataEntry*. Each entry represents a single time point, and contains channel voltage data as well as time-stamp and marker information.
- ❖ The *Classification* subfolder contains classes in the *MCAEmotiv.Classification* namespace. These classes implement classification algorithms as well as interfaces for interacting with them. Abstract classes like *AbstractClassifier* and *AbstractBinaryClassifier* are used to factor out common logic and make the *IClassifier* interface easier to implement.
- ❖ The *Analysis* subfolder contains classes in the *MCAEmotiv.Analysis* namespace. These classes provide useful preprocessing methods, such as a clean interface to the Math.Net FFT algorithm.

The *GUI* project implements the Experimenter application. It takes *Emotiv* as a dependency. The *GUI* code files are similarly divided into four subfolders, each of which corresponds to a namespace. The exception is the *Program* class, which acts as the main entry point for the application.

- ❖ The *Common* subfolder contains classes in the *MCAEmotiv.GUI* namespace. These classes typically provide utility and extension methods for manipulating and instantiating graphical components. The static *GUIUtils.GUIInvoker* is particularly important, as it provides a common and convenient way to run code on the user interface thread.
- ❖ The *Controls* subfolder contains classes in the *MCAEmotiv.GUI.Controls* namespace. These classes are all user controls which extend a default control class to implement specific components

of the Experimenter interface. Note that these controls were coded by hand instead of using Visual Studio's GUI designer, and thus cannot be opened in the designer. The reason for doing this is that controls created this way tend to be more maintainable and to behave better when the form is resized. Important classes in this namespace include the *ConfigurationPanel* and *DerivedTypeConfigurationPanel* controls which implement dynamic controls for configuring parameterized objects as well as the *MainForm* control which acts as the applications primary window.

- ❖ The *Configurations* subfolder contains classes in the *MCAEmotiv.GUI.Configurations* namespace. These classes act as parameterized models for the dynamic views created by *ConfigurationPanel*. While most of these classes are little more than lists of properties, the *StimulusClass* class also implements important functionality for loading stimulus classes from disk.
- ❖ The *Animation* subfolder contains classes in the *MCAEmotiv.GUI.Animation* namespace. These classes implement the yield/view presentation system. The *View* abstract class implements the underlying functionality for a view, the *IViewProvider* interface details the functionality which must be implemented by providers of views, and the *Animator* class implements the presentation loop for displaying views.

As its name suggests, the *Testing* project contains tests for many of the core classes in the *Emotiv* and *GUI* projects. These tests are contained in the *Unit* subfolder (*MCAEmotiv.Testing.Unit* namespace), and are invoked by the main *Program* class. Due to time constraints the test suite is by no means complete. However, it does validate many important features of classes, such as proper use of the *Serializable* attribute.