NASA/TM—2015-218824

Data Processing and Machine Learning Methods for Multi-Modal Operator State Classification Systems

Tristan A. Hearn
Glenn Research Center, Cleveland, Ohio

June 2015

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NASA Technical Report Server—Registered (NTRS Reg) and NASA Technical Report Server—Public (NTRS) thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counter-part of peer-reviewed formal professional papers, but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., "quick-release" reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at http://www.sti.nasa.gov

- E-mail your question to help@sti.nasa.gov

- Fax your question to the NASA STI Information Desk at 757-864-6500

- Telephone the NASA STI Information Desk at 757-864-9658

- Write to:
  NASA STI Program
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

NASA/TM—2015-218824

Data Processing and Machine Learning Methods for Multi-Modal Operator State Classification Systems

Tristan A. Hearn
Glenn Research Center, Cleveland, Ohio

National Aeronautics and
Space Administration

Glenn Research Center
Cleveland, Ohio 44135

June 2015

Level of Review: This material has been technically reviewed by technical management.

Available from

NASA STI Program
Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
703-605-6000

This report is available in electronic form at http://www.sti.nasa.gov/ and http://ntrs.nasa.gov/

Data Processing and Machine Learning Methods for Multi-Modal Operator State Classification Systems

Tristan A. Hearn
National Aeronautics and Space Administration
Glenn Research Center
Cleveland, Ohio 44135

Abstract

This document is intended as an introduction to a set of common signal processing and machine learning methods that may be used in the software portion of a functional crew state monitoring system. This includes overviews of both the theory of the methods involved, as well as examples of implementation. Practical considerations are discussed for implementing modular, flexible, and scalable processing and classification software for a multi-modal, multi-channel monitoring system. Example source code is also given for all of the discussed processing and classification methods.

Contents

1 Introduction

The Crew State Monitoring (CSM) Element is a task within the Vehicle Systems Safety Technologies (VSST) Project, which is part of the NASA Aviation Safety Program. The objective of the CSM Element is to develop technologies to assist in crew maintenance of appropriate readiness for and engagement in mission tasks by avoiding and detecting hazardous functional operator state. These measures include determination of attention (or lack thereof), task engagement, and workload. Such a system is being developed from a suite of neural, physiological and behavioral sensing modalities to an integrated multi-modal, multi-state system for in-task detection of hazardous operator state.

The integration of these technologies into a multi-modal state classification system necessitates careful planning of signal processing and selection of robust machine learning techniques for accurate and precise determination of cognitive states; ultimately under a real-time processing constraint. The purpose of this work is to document the practical considerations for working with data sources of these types, present the background theory of the machine learning systems of interest to perform the classification portion of for this system, and to illustrate strategies for the implementation of processing software in support of future simulator studies and the development of a functional crew state monitoring system.

This document is organized as follows. Section 1.1 describes how to setup the Python programming language, with the needed technical computing libraries. Section 2 discusses how raw data is acquired from selected hardware devices from software written in Python. These devices include the ISS Imagent fNIRS imaging system, Emotiv Epoc EEG headset, and Neulog brand sensor modules. Section 3 discusses various common signal processing functions which may be applied to raw fNIRS and EEG data (as examples). Section 4 describes several popular machine learning methods which may be used for cluster analysis or classification. Finally, Section 5 describes the top-level organization of software meant to implement acquisition, signal processing, and classifications functions involving multiple modalities with multiple channels.

1.1 Setting Up Python

Python is a high level interpreted programming language that has become very popular for scientific computing[26]. All example code within this document is written in Python, making use of the Numpy[1] package for array processing, the Scipy[2] package for scientific computing functions, the Matplotlib[3] package for plotting, and the scikit-learn[4] package for classification algorithms.

[1] http://www.numpy.org/
[2] http://www.scipy.org/
[3] http://matplotlib.org/
[4] http://scikit-learn.org/

1.1.1 Windows

The most straightforward method for setting up Python on Windows for scientific computing is to use a precompiled distribution, such as Python(x, y)[5], Anaconda distribution[6], or Enthought Canopy[7].

Each of these provide free single-point executable installers that include the standard interpreter for the Python programming language, a large variety of third-party scientific computing libraries (including each package listed above), as well as a suite of free compilers for C, C++, and Fortran.

1.1.2 Mac OSX

Mac OSX comes with a version of the Python programming language interpreter, however it is typically an outdated version that is difficult to install up-to-date third party libraries into. The best way to install an up-to-date version of Python is to use the Homebrew[8] package manager. First, install Homebrew according to the documentation provided on their home page. Then, open a terminal, and type the code shown in Listing 1.

Listing 1: Installing Python on Mac

```
brew install python
```

Next, Numpy, Scipy, Matplotlib, and Scikit-Learn can be installed by typing each line shown in Listing 2.

Listing 2: Installing Python libraries on Mac

```
pip install numpy
brew install gfortran
pip install scipy
pip install matplotlib
pip install scikit-learn
```

[5]https://code.google.com/p/pythonxy/
[6]https://store.continuum.io/cshop/anaconda/
[7]https://www.enthought.com/products/canopy/
[8]http://brew.sh/

1.1.3 Linux

Most popular desktop Linux distributions include an up-to-date version of the Python programming language interpreter, and the needed libraries are often available in the system's package manager. For example, if the Linux distribution uses the apt-get package management system, then the needed packages can be obtained by typing in each line shown in Listing 3 in a terminal window:

Listing 3: Installing Python libraries on Linux

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install python-numpy
sudo apt-get install python-scipy
sudo apt-get install python-matplotlib
sudo apt-get install python-sklearn
```

Any necessary dependencies (such as the gfortran compiler) will be downloaded and installed automatically from the distribution's remote repositories.

1.1.4 Alternatives To Python

While Python provides a very convenient development environment for technical computing, it is certainly not alone in this regard. MATLAB, for example, is based on a similairly easy-to-read scripting language, and has a large variety of array processing, scientific computing, and plotting functionality within its standard library. However, the many of the machine learning and general purpose model fitting methods discussed in this work exist within commercial add-on toolboxes, in particular the Machine Learning and Statistics Toolboxes. If desired, these methods may also be implementing by a user themselves at a low level.

GNU Octave, the R programming language, and the Julia programming are language other suitable choices for implementation of the described algorithms. These are each open source, and available free of charge.

2 Data Acquisition Software

The data collection for each modality is performed by a hardware device, which is interfaced to a computer system by some form of communication bus. Before any form of processing or classification may take place, the data must be collected from this interface and organized appropriately. This section discusses the collection of data from three separate instruments into a form that may be used in the Python programming language, in real time.

This is far from an exhaustive set of the types of devices one may be interested in using in an operator state classification system. However, these three examples are diverse enough to demonstrate how data may be collected from instruments which use similar protocols and computer interfaces in a laboratory environment.

2.1 ISS Imagent

The Imagent is a 16-channel frequency-domain fNIRS instrument made by ISS, Inc[9]. fNIRS is an emerging low-cost technique for measuring temporal changes in blood oxygen concentration at specific locations the brain[9]. Like fMRI, this type of measurement can be associated with brain function due to neurovascular coupling. fNIRS measurements have been shown to be consistent with measurements taken simultaneously from fMRI, with the benefit of being significantly more portable[12].

The Imagent connects to a computer using a specialty hardware interface. Data can be monitored in real time using the BOXY software system included with the device, which can be configured to transmit the data to a software serial port. By transmitting to a port that is being read by a different program, one can read and operate on the data within that program, in real time.

The data that is transmitted by BOXY may not necessarily include the calculated relative concentration changes of hemoglobin species, but rather they may be the raw optical density measurements of the device alone. This may be done so that the concentration change calculation may be implemented by the user directly, if this is preferred. Details of this calculation are shown in Section 3.2.

[9]http://www.iss.com/

Listing 4: BOXY Serial Output

```
A-C1=3.929E+0 A-C2=3.307E+0 A-C3=4.974E+2 A-C4=1.272E+3 A-C5=4.483E-1 A-C6
    =2.137E+0 A-C7=2.722E+1 A-C8=2.256E+2 D-C1=2.534E+2 D-C2=2.530E+2 D-C3
    =1.808E+3 D-C4=5.432E+3 D-C5=2.517E+2 D-C6=2.528E+2 D-C7=3.348E+2 D-C8
    =9.828E+2 P-C1=1.79859E+0 P-C2=5.15935E+1
...
```

BOXY writes data to serial ports in the format shown in Listing 4, as a sequence of channel names and corresponding channel values (with 4 digits of precision), with each name/value pair being separated with a single space. When the last channel value is written, a line break character is then written. A serial port parser can be written in Python to decode this pattern.

Listing 5: Reading ISS Imagent Data In Python

```
import serial

class Imagent(object):

    def __init__(self, port, baud):
        self.ser = serial.Serial(port=port,
                                 baudrate=baud)
        self.buffer = ''


    def read(self):
        raw_data = ser.read(ser.inWaiting())
        self.buffer += raw_data

        blocks = self.buffer.split("\n")

        values = []
        for block in blocks[:-1]:
            block_vals = []
            pairs = block.split()
            for pair in pairs:
                value = float(pair.split("=")[1])
                block_vals.append(value)
        self.buffer = blocks[-1]
        values.append(block_vals)
```

Listing 6: Usage Of The Imagent Class

```
from ISS import Imagent

device = Imagent('COM5', 115200)
while True:
    # acquires all fNIRS data available
    data = device.read()
    print data
```

Listing 5 shows a basic implementation of a program object in Python which can read data from an ISS Imagent that is streaming to a serial port. In this code, a software buffer is used to hold blocks of channel data which have only partially been written, to ensure that data is not needlessly lost. Data values for all channels are then collected by time blocks and returned. It is also fairly simple to modify this code to return the data as name/value pairs, if this is more desirable.

2.2 Emotiv Epoc

The Epoc is a 14 channel electroencephalography (EEG) headset made by Emotiv, Inc[10]. EEG is a recording of electrical activity measured on a head, activity which includes voltage fluctuations from neuron activations within the brain. For the study of brain function, it is often the magnitudes of repetitive sequences (frequency domain information) of neural activity that are of most interest[30].

The Epoc gathers data at a rate of 128 Hz, and transmits this data to a computer wirelessly via a provided USB Bluetooth adapter. The data can be monitored using the TestBench software included with the headset, which includes diagnostic utilities to detect bad channel connections. A software developer kit (SDK) is available to users who purchase a research use license. This SDK allows for direct programmatic access to the raw EEG data as it is acquired.

The SDK includes a dynamic linked library. This library will be the file "edk.dll" on Windows, "libedk.dylib" on Mac OSX, and "libedk.so" on Linux. This library may be called directly in order to communicate with the Epoc headset. Though the library was written and compiled using the C programming language, the library can be loaded and interacted with from Python if the functions contained within the library that a user wishes to interact with are known. The Emotiv SDK requires that you first initialize the device via the SDK's "EngineConnect()" function, and parse through a series of event states until the headset is ready to begin transmitting data. After the initialization is finished, data can be acquired. The basic usage of the Emotiv SDK from Python is shown in Listing 7.

[10]http://emotiv.com/

Listing 7: Emotiv SDK For Reading From An Epoc

```
from ctypes import c_char_p, c_uint, c_int,c_bool, CDLL
# Load DLL (Windows example)
edk = CDLL("edk.dll")

# initialize
connect_param = c_char_p(b'Emotiv Systems-5')
edk.EE_EngineConnect(connect_param)
data_handler = edk.EE_DataCreate()

# Set device buffer
edk.EE_DataSetBufferSizeInSec(5)

# Wait for data acquisition state
eEvent = self.edk.EE_EmoEngineEventCreate()
state = self.edk.EE_EngineGetNextEvent(eEvent)

while not state:
  state = self.edk.EE_EngineGetNextEvent(eEvent)
self.edk.EE_DataAcquisitionEnable(c_uint(0),c_bool(1))

# Now, read data samples from the headset's channels

nSamples = c_int()
while True:
    # determine number of samples available
    edk.EE_DataUpdateHandle(c_uint(0),
        data_handler)
    edk.EE_DataGetNumberOfSample(
        data_handler,byref(nSamples))
    n = nSamples.value

    # prep empty data structure
    container = np.empty((14 , n))
    for i in range(14):
        data = np.empty((1,n))
        data_ctype = np.ctypeslib.as_ctypes(
            data)
        edk.EE_DataGet(data_handler,
            i,byref(data_ctype),c_uint(n))
        data_read = np.ctypeslib.as_array(
            data_ctype)
        container[i,:] = data_read[0]

    print container
```

Listing 8: Emotiv Epoc Class

```
from ctypes import c_char_p, c_uint, c_int,c_bool
from ctypes import byref, CDLL
import numpy as np
import time, sys

class Epoc(object):
    """
```

```
    Class that connects to Emotiv Epoc by wrapping the
    research SDK dynamic link libraries
    """
    def __init__(self):
        #setup access to binaries
        if sys.platform=='darwin':
            edk_file='libedk.1.0.0.dylib'
        elif sys.platform=='win32':
            sys.path.append('lib')
            edk_file='edk.dll'
        self.edk=CDLL(edk_file)

        self.connected = False

    def connect(self, timeout = 10):
        """
        Establishes connection to Emotiv Epoc
        """
        connect_param = c_char_p(b'Emotiv Systems-5')
        self.edk.EE_EngineConnect(connect_param)
        self.data_handler = self.edk.EE_DataCreate()
        self.edk.EE_DataSetBufferSizeInSec(5)

        eEvent = self.edk.EE_EmoEngineEventCreate()
        state = self.edk.EE_EngineGetNextEvent(eEvent)
        t0 = time.time()
        while not self.connected:
          state = self.edk.EE_EngineGetNextEvent(eEvent)
          if not state:
            self.connected = True
            self.edk.EE_DataAcquisitionEnable(c_uint(0),
                c_bool(1))
            break

    def read(self):
        """
        Get block of raw data from the device buffer
        """
        nSamples = c_int()
        while True:
            self.edk.EE_DataUpdateHandle(c_uint(0),
                self.data_handler)
            self.edk.EE_DataGetNumberOfSample(
                self.data_handler,byref(nSamples))
            n = nSamples.value
            if not n:
                continue
            container = np.empty((14, n))
            for i in range(14):
                data = np.empty((1,n))
                data_ctype = np.ctypeslib.as_ctypes(
                    data)
                self.edk.EE_DataGet(self.data_handler,
                    i,byref(data_ctype),c_uint(n))
                data_read = np.ctypeslib.as_array(
                    data_ctype)
                container[i,:] = data_read[0]
```

```
            return container
```

This interface can be improved by implementing this basic initialization and data acquisition functionality into a class definition, for modular inclusion into a larger program. Listing 8 shows a basic implementation of a a program object in Python which can read data from an Emotiv Epoc, in real time. This class can be imported and used as shown in Listing 9. This class was designed to operate with an interface comparable to the class used to read data from the ISS Imagent.

Listing 9: Using the Emotiv Epoc Python Class

```
from emotiv import Epoc

device = Epoc()
while True:
    # acquires all EEG data currently available
    data = device.read()
    print data
```

2.3 Neulog brand sensors

Neulog[11] is a brand of sensors made by Scientific Educational Systems, Ltd. These sensors are designed as individual hardware modules, which may be connected together using a common interface. These blocks include a number of low cost, portable, single-channel sensor technologies, such as galvanic skin response (GSR) and plethysmograph-based heart rate estimation.

Blocks of these connected sensors are then ultimately interfaced to a computer using a USB cable or via a wireless connection. Software is provided to monitor and record the data collected by the sensors. The device itself is seen by the software as a serial device, which allows for other programs to access the data directly by using the same protocol used by the provided software.

Listing 10: Neulog Sensors Class

```
import serial
import time
import os

class Neulog(object):
    def __init__(self, port, baud):
        self.ser = serial.Serial(port=port,
```

[11] http://www.neulog.com/

```
                                baudrate=baud)
        self.status = 'connected'
        self.buf = []
        t = time.time()
        while not self.connect():
            if time.time() - t > 2:
                break

    def send(self, s, checksum = False):
        time.sleep(0.02)
        self.ser.flushInput()
        self.ser.flushOutput()
        for c in s:
            self.ser.write(c)
        if checksum:
            self.ser.write(chr(sum([ord(c)
                for c in s]) % 256))

    def receive(self, i = False):
        time.sleep(0.02)
        iw = self.inWaiting()
        if False == i: i = iw
        if iw >= i:
            r = self.read(i)
            return r
        return 'False'

    def connect(self):
        self.ser.close()
        self.ser.open()
        self.ser.send(chr(85) + 'NeuLog!')
        if 'OK-V' != self.receive(4): return False
        self.status = 'connected'
        return '.'.join([str(ord(c))
            for c in self.ser.receive(3)])

    def scanStart(self):
        if self.status != 'connected': return False
        self.send(chr(18) + chr(96) + \
             chr(34) + chr(9), True)
        r = self.receive(4)
        print "What's this: %i" % (ord(r[-1]))
        if chr(18) + chr(96) + chr(11) == r[:-1]:
            self.status = 'scanning'
            return True
        return False

    def scanRead(self):
        if self.status != 'scanning':
            return False
        sensors = []
        r = self.receive()
        while len(r) > 7:
            chunk, r = r[:8], r[8:]
            if chr(85) != chunk[0]:
                continue
            chunk = [ord(c) for c in chunk]
```

```
            check = chunk[-1] != sum(chunk[:-1]) % 256
            if check:
                continue
            stype, sid, ssndver = chunk[1:4]
            sver = '.'.join([str(i)
                for i in chunk[4:7]])
            sensors.append((stype, sid, sver))
        return sensors

    def scan(self):
        t = time.time()
        sensors = []
        self.scanStart()
        time.sleep(1)
        sensor = self.scanRead()
        while len(sensor) !=0:
            sensors += sensor
            sensor = self.scanRead()
        self.scanStop()
        self.sensors = sensors


    def scanStop(self):
        if self.status != 'scanning':
            return False
        self.send(chr(18))
        self.receive()
        self.status = 'connected'
        return True

    def getSensorsData(self, stype, sid):
        if self.status != 'connected':
            return False
        self.send(chr(85) + chr(stype) + \
            chr(sid) + chr(49) + (3 * chr(0)), True)
        r = self.receive()
        if not r or chr(85) != r[0] or chr(49) != r[3]:
           return False
        r = [ord(c) for c in r]
        if r[-1] != sum(r[:-1]) % 256:
            return False
        return r

    def read(self):
        data = []
        for stype, sid, vid in self.sensors:
            x = self.device.getSensorsData(stype,sid)
            data.append(x)
        return data
```

Listing 11: Using the Neulog Python Class

```
from Neulog import Neulog

device = Neulog("COM3", 9600)

# Determine which sensors are connected
# (GSR, Heart rate, etc.)
device.scan()

while True:
    # acquires all sensor data currently available
    data = device.read()
    print data
```

3 Data Processing Methods

In this section, common signal processing functions which may be used for each modality of interest are discussed, and strategies for implementation in a real-time setting are presented. Example source code is also given for each processing method. Note that the organization of each processing method under a listed modality should not suggest that a particular method is exclusive to that particular modality. Most of the discussed processing methods are generally applicable for a large variety of purposes.

3.1 EEG Data Processing

Processing methods for EEG data largely take place in frequency domain (within the transform domain of the discrete Fourier transform or a similar orthonormal transformation), though not all EEG processing methods are necessarily of this type.

EEG devices have a fast sample rate compared to fNIRS. For example, the Emotiv Epoc collects data at the rate of 128 Hz, compared to the ISS Imagent at 6 Hz. This necessitates that the selection of the data processing functions of a real-time EEG monitoring system must take into account computational complexity, to prevent significant lag between acquisition and classification. While this is still true of an fNIRS monitoring system, it is less of a concern due to the lower sampling rate than for an EEG monitoring system.

If the sample rate is much higher than what is necessary to monitor all of the phenomena of interest, then it is also possible to down-sample the data during acquisition (ie. collect every other data point rather than the entire data buffer), to reduce the dimension of the raw data to only what is needed in processing.

3.1.1 Frequency Band Filtering

If a set of frequency bands of interest are known a priori, it is possible to remove any other frequency bands from the data using band-pass filtering using the discrete Fourier transform, typically via an implementation of Fast Fourier Transform (FFT). The j^{th} component of the discrete Fourier transform $\hat{F}$ of a one-dimensional signal F is given as

$$\hat{F}_j = \sum_{k=1}^{n} \exp\left(\frac{-2i\pi jk}{n}\right) F_k.$$

Recalling the Euler formula $\exp(i\theta) = \cos(\theta) + i\sin(\theta)$, it is clear that the discrete Fourier transform fits a series of sine and cosine of increasing frequency to the input data. By analyzing or manipulating the values of $\hat{F}_j$, the contribution of each fundamental sine and cosine frequency can be quantified, or even modified artificially.

This form of filtering may be explicit (ie. we perform the transform, manipulate the coefficients in transform domain within the band of interest, then invert the transformation back into time domain), or implicit (analysis is conducted in frequency domain within a band of interest, but the transform is not inverted).

For instance, if we would like to compute the spectral power of α waves (which operate between 8 Hz and 15 Hz) within a block of EEG data from a channel, then we must first compute the FFT of the data, isolate the coefficients in transform domain corresponding to this frequency band, and return the sum of the square magnitudes of these coefficients. A Python implementation of this is shown in Listing 12, with real-time computation of α and β waves within a stream of data from an Emotiv Epoc. A general-purpose spectral power function is implemented, that accepts raw data and band specification as input in order to compute the spectral power within that band. This is then used to implement functions to compute the power within the α wave and β wave spectral bands. Note that if the power within a number of spectral bands is needed within production code, it would be more efficient to implement a function to compute them with only a single FFT.

Listing 12: Computing α And β Wave Levels In Python

```
import numpy as np
from emotiv import Epoc

def band_power(data, sample_rate, band):
    """
    General function for computing
    spectral power
    """
    N = data.shape[1]
    windowed = np.hamming(N)*data
    freqs = float(sample_rate)/N*np.arange(N/2 + 1)
    psd = np.abs(np.fft.rfft(windowed, axis = 1))**2


    idx = np.sum(psd[:, (freqs >= band[0])
          & (freqs <= band[1])], axis=1)

    return idx

def alpha_levels(data, sample_rate=128):
    return band_power(data, sample_rate, [8, 15])

def beta_levels(data, sample_rate=128):
    return band_power(data, sample_rate, [16, 31])


device = Epoc()
data = device.read()
buffer_size = 1024
while True:

    new = device.read()
    data = np.concatenate((data, new), axis  = 1)

    if data.shape[1] > buffer_size:
        data = data[:,-buffer_size:]

    alpha = alpha_levels(data)
    beta = beta_levels(data)
    print "Current alpha wave level:", alpha
    print "Current beta wave level:", beta
```

3.1.2 Matched Filtering

In some cases, it is desirable to remove noise from EEG data which may be better thought of as time-domain phenomena. For instance, in the case of a short-lived artifact (such as an eye blink or other momentary muscle twitch), the contamination may be momentary, and perhaps does not occur at any predictable frequency. Band-pass filtering would be ill-suited to remove such an artifact. Instead, a process known as matched filtering may be applied to

remove these artifacts.

In matched filtering, the cross-correlation function (the convolution between a discrete function and the time-reversal of another discrete function) of a data source and a template function is computed in order to identify instances of the template within the data. By selecting a template which represents a form of contamination (such as an eye blink) and identifying locations where this contamination occurs, localized filtering may be applied to remove the contamination from the data.

Thanks to the Fourier convolution theorem, cross-correlations can be efficiently computed in $O(n \log n)$ operations, using the FFT[3]. Thus, if an FFT implementation is available, matched filtering is relatively simple to implement.

Listing 13 shows a simple implementation of a matched filter for the reduction of blink artifacts in a single channel of EEG data, shown at top of Figure 2. An FFT-based convolution function is available within the SciPy library, and was used to compute the cross-correlation function. For this example, a blink template (shown in Figure 1) was collected from another data set, and the cross-correlation between it and the EEG channel was computed (shown in the middle plot of Figure 2). The isolated peaks within the cross-correlation indicate time locations within the EEG channel which correlate very highly with the blink template. By subtracting off a multiple of the template from the channel data only at these maximum-correlation locations, the blink artifacts may be removed, as seen in the bottom plot of Figure 2.

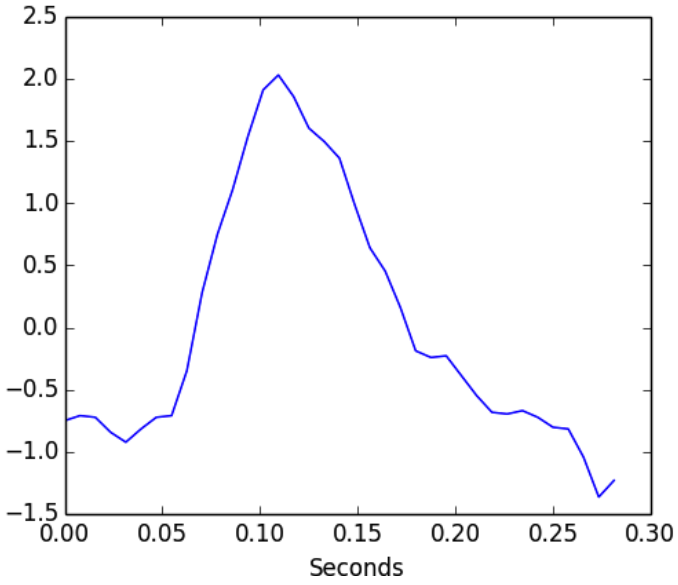


Figure 1: Eye blink artifact template used for matched filtering.

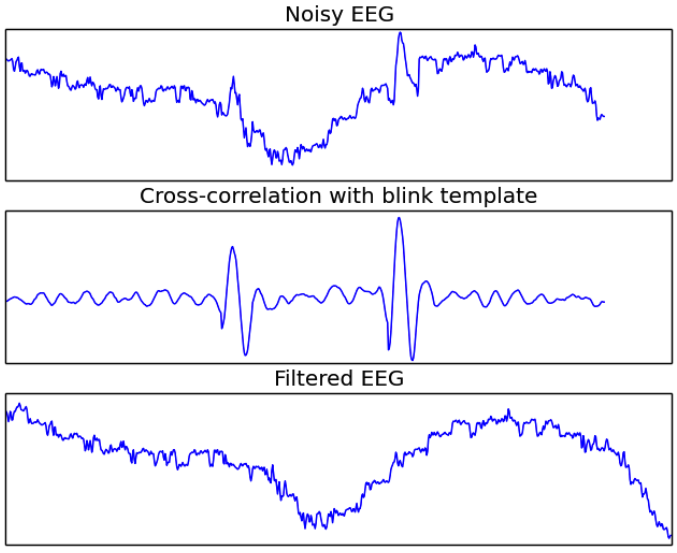


Figure 2: Example of matched filtering for the reduction of eye blink artifacts.

Listing 13: Computing A Matched Filter In Python

```
import numpy as np
from scipy.signal import fftconvolve

# Load a noisy EEG channel
signal = np.loadtxt("eeg_noisy.dat")

# Load saved blink template
blink = np.loadtxt("blink.dat")

# Cross-correlation is convolution
# with a reversed kernel
cross_corr = fftconvolve(signal, blink[::-1],
                         mode="same")

# find peaks, and subtract template:
filtered = np.zeros(signal.shape)
filtered[:] = signal[:]
filtered[300:337] += -15*blink
filtered[500:537] += -20*blink
```

3.1.3 Wavelet Filtering

One weakness of matched filtering is that positive detection does require that an instance of the template to be matched within a signal has a magnitude comparable to that of the template itself. If it is desired to perform detection and filtering in the case where a template is known, but the location and amplitude are not, then wavelet threshold filters may be an appropriate filtering method. Wavelet filters are also a very effective method for removing unwanted noise and slow drift within data.

The discrete wavelet transform has become a popular tool for the compression, denoising, and general analysis of single and multidimensional digital signals. There are a number of inherent features of the discrete wavelet transform which motivate its use for these and other purposes[21]. In simplest terms, the discrete wavelet transform is a simultaneous decomposition of a signal in both time (or space) and frequency. It is most often computed by successive convolution with a set of digital filter banks. This form of the discrete wavelet transformation is known as Multi-Resolution Analysis (MRA).

Given a discrete signal $f \in \mathbb{R}^m$ of dyadic dimension $m = 2^k, k \in \mathbb{Z}$, the decomposition of f into wavelet coefficients (written as $\widehat{f}$) will have a dual subband structure[21]. The first half of the components of $\widehat{f}$ will

represent a low pass filtered version of f, and the remaining components will represent a high pass filtered version of f. These subbands are also referred to as approximation coefficients and detail coefficients, respectively. These subbands are generated through discrete convolution with a scaling filter $\phi[x] \in \mathbb{R}^m$ and wavelet filter $\psi[x] \in \mathbb{R}^m$, respectively. So at the first level, the two subbands H_1 and L_1 are computed as

$$L_1 = \underset{2}{\downarrow}[f * \phi], \quad H_1 = \underset{2}{\downarrow}[f * \psi] \ .$$

The $*$ represents discrete convolution, and $\underset{2}{\downarrow}$ represents the dyadic down sampling operation, which is performed by discarding every other component in the vector upon which it is applied. Finally, the two down sampled subbands are combined into single vector so that $\widehat{f} \in \mathbb{R}^m$.

This combination of discrete convolution and down sampling to produce an MRA represents an overall linear transformation[21]. Therefore, a 1-level discrete wavelet transform may be written as

$$\widehat{f} = \mathcal{W}_{\phi,\psi} f = \begin{bmatrix} L_1 & H_1 \end{bmatrix}^T,$$

where $\mathcal{W}_{\phi,\psi} \in \mathbb{R}^{m \times m}$ is a matrix encoding the discrete wavelet transform.

In the case where wavelet and scaling filters ψ and ϕ are chosen to produce an orthonormal discrete wavelet transform, it would be the case that $\mathcal{W}_{\phi,\psi}^T \mathcal{W}_{\phi,\psi} = \mathcal{W}_{\phi,\psi} \mathcal{W}_{\phi,\psi}{}^T = f$. Thus the inverse wavelet transform would be given by

$$f = \mathcal{W}_{\phi,\psi}^T \widehat{f}.$$

In the case of biorthogonal wavelet transforms, $\mathcal{W}_{\phi,\psi}$ will fail to be an orthogonal matrix. However, its inverse will still be well-defined, so that

$$f = \mathcal{W}_{\phi,\psi}^{-1} \widehat{f}.$$

To compute a successive level of the decomposition, the exact procedure described above is repeated on the low pass subband (the approximation coefficients) L_1 of $\widehat{f}$. Thus the multi-level 1D forward and inverse discrete wavelet transforms are generated recursively through successive applications of single level wavelet transformations. A j-level 1D discrete wavelet transform can be expressed as a single linear transformation

$$\widehat{f} = \mathcal{W}_{\phi,\psi,j} f = \begin{bmatrix} L_j & H_j & \cdots & H_1 \end{bmatrix}^T,$$

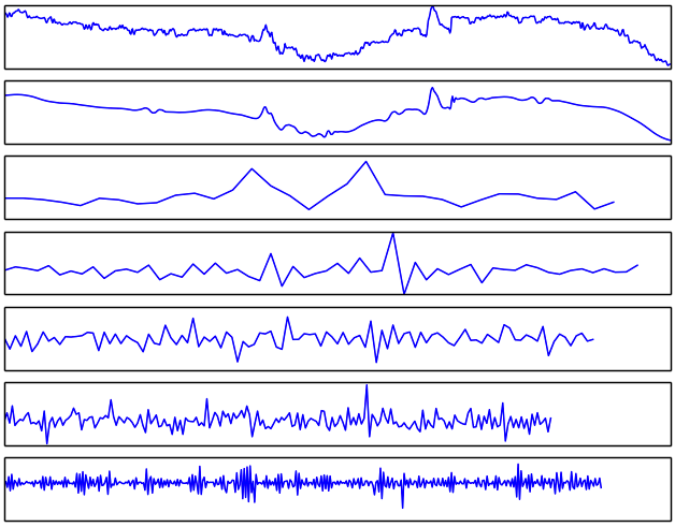


Figure 3: Four level wavelet transformation of single-channel EEG data. The top plot is the original data, all others are the wavelet coefficients at each decomposition level.

with orthogonality and biorthoginality leading to well-defined inverse transformations, just as in the case of a one-level transformation. The computational complexity of the 1D discrete wavelet transform is $O(n)$, where n is the length of the input.

Filtering methods based on the discrete wavelet transform operate by choosing some $\tau_j > 0$ for shrinking the magnitude of the coefficients within the detail subbands of the j levels. Ultimately, the use of wavelet filtering requires the selection of a type of wavelet filter (which determines the pair (ϕ, ψ)), the number of decomposition levels to perform, and the threshold values to apply. The Pywavelets[12] package in Python implements a variety of discrete wavelet transformation functions and helper methods.

[12]http://www.pybytes.com/pywavelets/

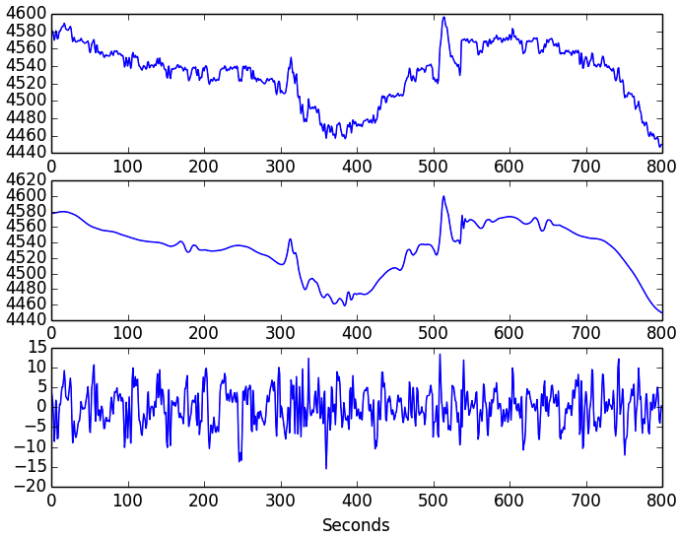


Figure 4: Artifact and slow drift removal of an EEG channel using wavelet filtering. The original EEG channel is the top plot, the wavelet reconstruction of slow drift and blink artifacts is the middle plot, and their difference gives the bottom plot.

Listing 14: Using A Wavelet Filter In Python

```
import numpy as np
import pywt

# Load a noisy EEG channel
signal = np.loadtxt("eeg_noisy.dat")

# Compute a 5-level discrete wavelet transform
# using the Db5 wavelet filter
coeffs = pywt.wavedec(signal, "db5",level=5)

# Filter the coefficients to separate out
# high-frequency data from low-frequency
# data, at multiple levels of resolution
c = [coeffs[0]]
for i in xrange(5):
    new_coeff = pywt.thresholding.hard(coeffs[i+1],20)
    c.append(new_coeff)

# construct the artifact+drift approximant
filtered = pywt.waverec(c, "db5")

# remove slow drift from the signal
signal = signal - filtered
```

3.2 fNIRS Data Processing

There are a variety of ways in which fNIRS data can be processed, which is largely due to the number of hardware implementations of fNIRS technology[8, 17]. Continuous-wave fNIRS is the simplest implementation of the technology, and the associated calculations can be applied to data taken from the ISS Imagent.

3.2.1 Modified Beer-Lambert Law

The most basic data processing function to be implemented is based off of the Modified Beer-Lambert Law (MBLL), which quantifies the relationship between relative changes in the concentrations of oxygenated and deoxygenated hemoglobin ($\Delta[HbO]$ and $\Delta[Hb]$, respectively) and the optical intensity measurements taken by the Imagent[5]. The MBLL has the form

$$\begin{pmatrix} \Delta[HbO] \\ \Delta[Hb] \end{pmatrix} = \begin{pmatrix} \epsilon_{HbO,690} & \epsilon_{Hb,690} \\ \epsilon_{HbO,830} & \epsilon_{Hb,830} \end{pmatrix}^{-1} \begin{pmatrix} \mu_{690} \\ \mu_{830} \end{pmatrix}$$

where $\epsilon_{i,k}$ is the extinction coefficient of species i at wavelength k, and μ is computed as

$$\mu_i = \frac{\log\left(\frac{-I_{t_0,i}}{I_{t,i}}\right)}{r * DPF_i}$$

where DPF_i is the differential path length factor at wavelength i, $I_{t_0,i}$ is a baseline optical intensity measurement taken by the fNIRS instrument as a time $t = 0$ for wavelength i, $I_{t,i}$ is the current optical intensity measurement at wavelength i, and r is the source-detector separation for the current measurement[5, 31]. These are parameters which should have known values prior to the data processing.

Listing 15: Computing the MBLL in Python

```
from math import log
from ISS import Imagent

def mu(I, r, dpf, baseline):
    return log(abs(baseline/I))/(r*dpf)

def MBLL(i690,i830,baseline690, baseline830):
    # These constants are specific to
    # an experimental setup
    e_hbo_l1 = 0.956
    e_hbo_l2 = 2.3153
    e_hb_l1 = 4.9307
    e_hb_l2 = 1.7914
    dpf1,dpf2 = 5.49, 6.0
    r = 1.25

    mu_l1 = mu(i690,r,dpf1,baseline690)

    mu_l2 = mu(i830,r,dpf2,baseline830)

    denom = e_hb_l1*e_hbo_l2 - e_hbo_l1*e_hb_l2

    hbo = (e_hb_l1*mu_l2 - e_hb_l2*mu_l1)/denom

    hb = (e_hbo_l2*mu_l1 - e_hbo_l1*mu_l2)/denom

    return hb, hbo

# initialize data acquisition
device = Imagent('COM5', 115200)
# get baseline measurements
base1, base2 = device.read()

while True:
    # get new data
    i690, i830 = device.read()
    print MBLL(i690, i830, base1, base2)
```

Listing 15 shows a Python implementation of the MBLL, involving acquisition and basic real-time processing of a single channel fNIRS data stream from the ISS Imagent. The data is collected using the code that was shown in Section 2.1.

3.2.2 Physiological Corrections

fNIRS data may contain a significant amount of physiological noise[20]. If the physiological noise is limited to particular spectral bands, then band-pass or wavelet filtering are potential methods for the removal of this kind of noise (see Section 3.1.1 for details and implementations).

Another option is to collect an additional channel of fNIRS data at a depth shallower than brain tissue monitored by the primary channel. This channel can then be used to remove physiological noise via subtraction or regression[10].

If the data is to be processed offline, and it is believed that a physiological signal is present within a data source which is independent from the signal that one wishes to isolate, then Principal or Independent Component Analysis are also potential filtering strategies (see Sections 3.3.1 and 3.3.2).

3.3 Other General-Purpose Processing Methods

There are a large variety of signal processing methods and general purpose transformations which may be of use in processing and analyzing data, depending on the context.

3.3.1 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a procedure which seeks to re-express a collection of data in a more meaningful way, in the sense of reducing noise, removing redundancy or revealing hidden dynamics and unknown coupling relationships between data sources.

More specifically, PCA is an orthogonal transformation P on a matrix of data sources $A \in \mathbb{R}^{m \times n}$, designed so that the columns of $Y = P \cdot A$ are numerically decorrelated, in the sense that

$$\sigma^2_{Y_i,Y_k} = \frac{1}{n-1}\left(Y_i - \bar{Y_i}\right)\left(Y_k - \bar{Y_k}\right)^T = 0,$$

where $i \neq k$ and $\bar{Y_i}$ and $\bar{Y_k}$ are the mean values of Y_i and Y_k, respectively. The rows of the matrix P are referred to as the principal components. In order for each column vector in the new basis to be uncorrelated, the covariance matrix S_Y of $Y = PA$ must be diagonal

$$S_Y = \frac{1}{n-1}\bar{Y}\bar{Y}^T = \begin{pmatrix} \sigma^2_{Y_1} & & 0 \\ & \ddots & \\ 0 & & \sigma^2_{Y_n} \end{pmatrix},$$

where $\bar{Y}$ is the matrix Y with the mean of each column subtracted off, and the diagonal elements of S_Y are the individual variances of the columns Y. This

can also be written in terms of the original matrix A and the transformation matrix P

$$S_Y = \frac{1}{n-1}\bar{Y}\bar{Y}^T = P\left(\frac{1}{n-1}AA^T\right)P^T,$$

where one quickly notes that $\frac{1}{n-1}AA^T$ is a real symmetric matrix. Recalling that every real symmetric matrix is diagonalized by an orthonormal matrix of its own eigenvectors, it follows that by normalizing the matrix A and getting the eigenvectors of $\frac{1}{n-1}AA^T$ gives the principal components directly. Rather than approach this problem by directly computing the eigenvectors of $\frac{1}{n-1}AA^T$, it is much more effective and numerically stable to compute them by way of the Singular Value Decomposition (SVD).

The SVD is a matrix decomposition of the form

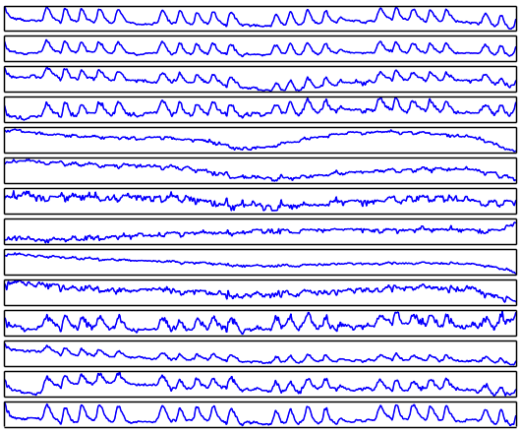
$$A = U\Sigma V^T,$$

where $U \in \mathbb{R}^{m\times m}$ is a orthonormal matrix of the eigenvectors of AA^T, $V \in \mathbb{R}^{n\times n}$ is an orthonormal matrix of eigenvectors of A^TA, and $\Sigma \in \mathbb{R}^{m\times n}$ is a diagonal matrix containing the square roots of the eigenvalues of AA^T along its main diagonal. The SVD has a number of efficient and stable implementations which are more effective than forming AA^T directly.

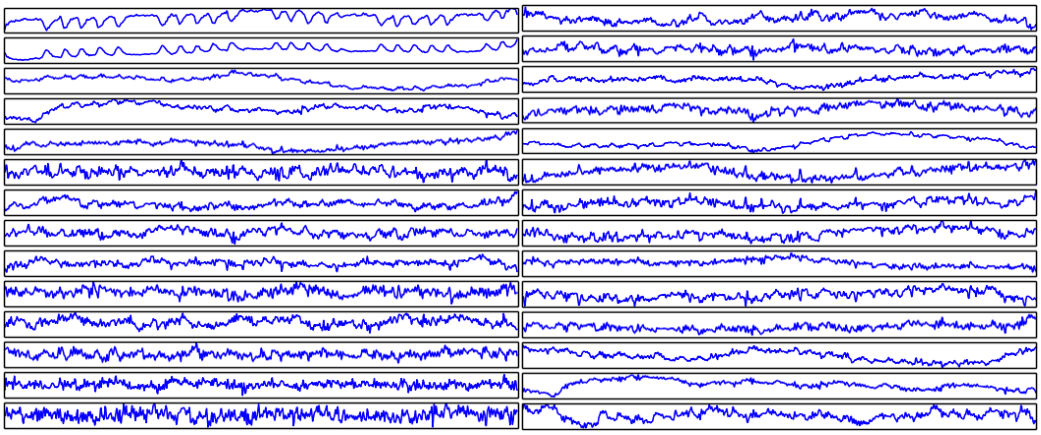
Since the PCA transformation matrix is orthonormal, the transform can always be easily inverted by multiplying by P^T (the transpose of one of the matrix of eigenvectors recovered using the SVD). In this way, any manipulations made within PCA transform domain may be manifested as filters within the original data.

As an example of this, consider the 7 second block of 14-channel EEG data shown in Figure 5(a), which is contaminated with a series of rapid eye blink artifacts in nearly half of the channels. In this case, each channel of EEG data is stored as a row within a matrix, and the PCA of this matrix is shown in Figure 5(b), in which the eye blinks are largely concentrated within the first two principal components. By multiplying these two components by zero and inverting the PCA, we retrieve a filtered version of the original data, with the eye blink artifacts removed (Figure 5(c)).

Since the SVD exists for all finite dimensional matrices and can be computed in a stable manner, PCA can be peformed for all but the very largest data sets.



(a) Original data



(b) PCA of data (c) Filtered result

Figure 5: Graphical example of PCA as a filtering method, for removing rapid eye blink artifacts.

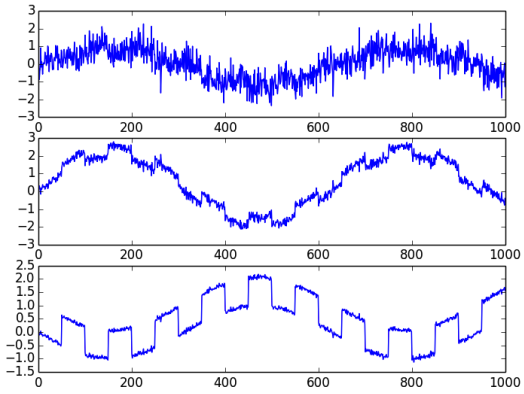
3.3.2 Independent Component Analysis (ICA)

Independent Component Analysis (ICA) is family of algorithms which are each very similar in structure and purpose to PCA. ICA also performs a linear transformation on a given matrix of data to produce a new matrix of data, but instead of defining a transformation so that the resulting vectors are decorrelated, ICA seeks to maximize higher order measures of statistical independence, such as negative kurtosis and mutual entropy[13]. The various independence measures that may be approximated are what largely differentiate the various ICA implementations that exist.

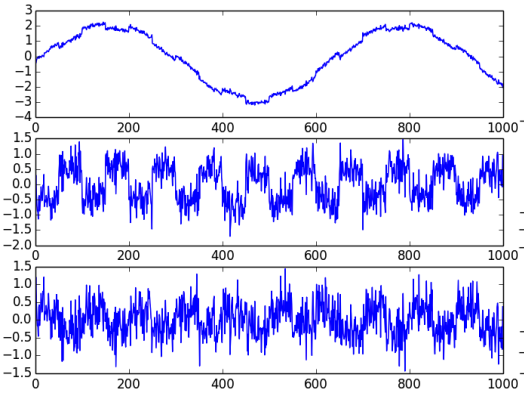
Compared to PCA, ICA can often separate independent sources and perform filtering more effectively, as the transformation is not restricted to be orthogonal. However, ICA algorithms may not necessarily converge for a given data set, and thus may not always be applied for a given problem.

An example of ICA is shown in Figure 6. In this example, a sine wave, a square wave, and a random noise source were added together with different ratios to produce 3 mixed data sources, shown in Figure 6(a). Applying PCA to them, shown in Figure 6(b) does manage to separate out the sine wave somewhat, though not very effectively. However, applying ICA very effectively separates out the three sources from the mixtures, as seen in Figure 6(c).

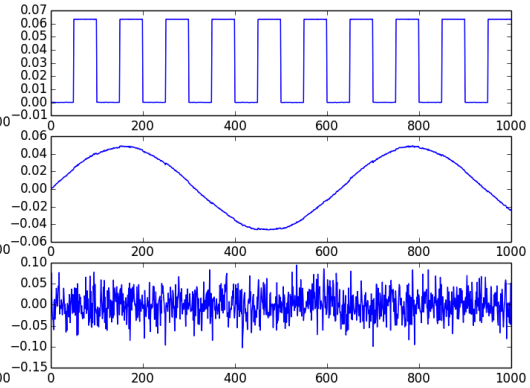
Listing 16 shows how to use PCA and ICA in Python, using the scikit-learn library.



(a) Original data



(b) PCA of data



(c) ICA of data

Figure 6: Graphical example comparing PCA and ICA for the separation of mixed signals. This original data (a) was created by mixing different multiples of a sine wave, square wave, and random noise source, which are not very well separated by PCA(b) but are very effectively recovered by ICA.(c)

Listing 16: Using PCA and ICA In Python

```
import numpy as np
from scipy import signal
from sklearn import decomposition

# Create example signals
x = np.arange(0,10,0.01)
sin_wave = np.sin(x)
noise = 0.5*np.random.randn(len(x))
square = signal.square(10*x)

# Make mixtures
mix_1 = sin_wave + noise - square
mix_2 = 2*sin_wave - 0.3*noise + 0.75*square
mix_3 = -sin_wave + 0.1*noise + square
mixed_data = np.array([mix_1, mix_2, mix_3]).T

# Compute PCA to recover signals
ica = decomposition.FastICA()
ica_data = ica.fit_transform(mixed_data)

# Compute ICA to recover signals
pca = decomposition.PCA()
pca_data = pca.fit_transform(mixed_data)

# Get ICA transformation matrix, P
P = ica.get_mixing_matrix()

# Use already fitted ICA tranform on some new data
new_data = np.random.randn(100,3)
new_ica_data = ica.transform(new_data)
```

PCA, ICA, and other matrix decomposition algorithms may be too computationally expensive to use in a real-time system (in the sense of recomputing the decomposition at each iteration of a real-time loop). However, if a transformation P associated with a decomposition like PCA or ICA is pre-computed before operation, it is very possible to apply this transformation to new acquired data sets in a online system, if it is believed that the dynamics between the sources which are being separated are constant over time. This is demonstrated at the bottom of Listing 16.

4 Machine Learning Methods

There are a variety of machine learning algorithms that are commonly used for the analysis, representation, and generalization of data sets. These include methods for finding unknown patterns within given data (graphically or numerically), or exploiting known patterns in data in order to provide needed functionality in a system. Broadly speaking, we are interested in two types of algorithms: Classification, Clustering.

Clustering is an exploratory data mining process that seeks to partition a set of observations into groups (or "clusters"), such that observations which belong to the same cluster are more similar to each other (with respect to some measurement) than they are to observations within different groups[32].

Classification is the process of determining which among a set of categories that an observation belongs to, using some characteristics of the observation together with any known information about the considered categories. One example of a classification system would be a "spam" filter for an e-mail server, where a determination must be made by the server whether a new received message should be delivered to a recipient or not. Classification methods typically require a "training" phase, where the system is provided with sets of example data that has been pre-classified into the categories of interest, so that a robust set of decision rules may be determined. For the e-mail filter example, this would involve selecting a set of valid e-mails along with a set of unsolicited spam e-mails, and providing them both to the classification system so that it may be trained appropriately.

The following sections discuss a few selected classification and clustering methods.

4.1 Naive Bayes Classification

Naive Bayes is a simple classification algorithm that is based on Bayes' theorem of conditional probability. It is simple to formulate, and is one of the few classification methods which (by its definition) attaches empirical probabilities to its classification results. However, its use is predicated upon certain assumptions which may not necessarily generalize well to more complex problems involving highly non-linear couplings between data measurements. Principally, the assumption of independence between the separate features used for classification. The "Naive" part of the algorithm's name is due to this assumption of independence, though this alone should not discourage

its use in practice[33]. Naive Bayes classification has been used successfully for problems such as e-mail spam filtering[29], general text classification[22], and cognitive state classification [24, 19].

Naive Bayes Classification can be derived as follows. Suppose we would like to find the probability of a certain discrete category $y = \{0, 1, \ldots, n\}$ based on a collection of data measurements $X \in \mathbb{R}^m$ (for instance, determining whether a medical patient has a particular illness or not (y_0 or y_1), based upon a collection of $x_1, x_2, \ldots, x_m$ separate measurable risk factors). Using Bayes' theorem of conditional probability, it would follow that

$$P(y_0|X) = \frac{P(y_0)\,P(X|y_0)}{P(X)}$$

$$P(y_1|X) = \frac{P(y_1)\,P(X|y_1)}{P(X)}$$

$$\vdots$$

$$P(y_n|X) = \frac{P(y_n)\,P(X|y_n)}{P(X)}$$

where $P(y_i|X)$ (the value of interest) is the probability of state y_i given data X, $P(y_i)$ is the overall probability of state y_i, $P(X|y_i)$ is the probability of the values of the data assuming y_i is the current state (similar to the definition of a $p-$value[7] in statistical hypothesis testing), and $P(X)$ is the probability of the measured data overall. If all of these values can be computed, then the value of $P(y_i|X)$ which has the highest value indicates the most likely state.

Now, if we assume that each data measurements (or "features") $X = [x_1, \ldots, x_m]$ is independent, then it follows that their joint probability is equal to the product of their marginal probabilities, so that

$$\begin{aligned} P(X|y_i) &= P(y_i)\,P(x_1|y_i)P(x_2|y_i) \cdot \ldots \cdot P(x_m|y_i) \\ &= P(y_i) \prod_{k=1}^{m} P(x_k|y_i). \end{aligned}$$

Therefore, the relationship between each type of feature and each state ($P(x_1|y_i)$) can each be summarized separately as an expectation over a prob-

ability distribution. The most common practice is to select a type of distribution for each one of these features, and then fit the free parameters of this distribution using some known data, using the common least-squares or maximum-likelihood estimation methods. For example, one popular choice is the Gaussian distribution, which gives

$$P(x_k|y_i) = \int_{\mathbb{R}} x_k \left(2\pi\sigma_{i,k}^2\right)^{-\frac{1}{2}} \exp -\frac{\left(x_k - \mu_{i,k}\right)^2}{2\sigma_{i,k}^2} dx_k.$$

Fitting a naive Bayes classifier would then involve individually fitting μ and σ values for each one of these $k \cdot i$ distributions. This may be computationally expensive depending on the amount of data that is available and selected to fit the models, but not difficult to implement, as each represents a standard problem in statistical point estimation.

In contrast, the factors $P\left(y_i\right)$ are determined a priori either from available theory (or are perhaps left as tuning parameters), and the values $P\left(X\right)$ in the denominator are most often computed implicitly as normalizing constants, after all other values have been computed.

Listing 17 shows how to use a Naive Bayes classifier with Gaussian likelihood models in Python, using the scikit-learn library. In this code, a base class is imported and used which automatically fits all of the μ and σ parameters, and estimates the $P\left(y_i\right)$ factors based upon the distribution of states present within the training data. This program object can then be used to classify new data directly, measure the probability of individual states, and more.

Listing 17: Using A Naive Bayes Classifier In Python

```
from sklearn import datasets
from sklearn.naive_bayes import GaussianNB

# Collection of data
# (four features per data point)
data = [[0,11,3,10],
        [10,11,1,0],
        [10,11,13,11],
        [1,11,0,11],
        [10,11,2,1],
        [10,11,13,11]]

# The state associated with each of the
# above data points
states = [0,1,2,0,1,2]

# Create a naive bayes classifier w/ gaussian
# likelihood models
gnb = GaussianNB()

# Fit to our data
gnb.fit(data, states)

# Measure its accuracy
print (states != gnb.predict(data)).sum()

# Show state probabilites for a new point
probabilities = gnb.predict_proba([1,8,0,8])

# Classify a new point (simply return state with
# the highest probability)
result = gnb.predict([1,8,0,8])
```

4.2 Support Vector Machines (SVMs)

Support Vector Machines (SVMs) are a collection of machine learning algorithms which can be used to perform classification or regression. They were introduced by Guyon and Vapnik in 1995[4], and have since been become widely used in a number of distinct problem domains, from text processing[14], to face detection in digital images[27], classification of brain states in fMRI[25] and EEG[6], and beyond.

As a classification algorithm, SVMs require a set of pre-classified training data to be provided, which is then used to fit an appropriate classification model. This model is then used to assign categories to future data.

More generally, say we have n observations of data available, with each observation made up of m separate values, and each observation known to

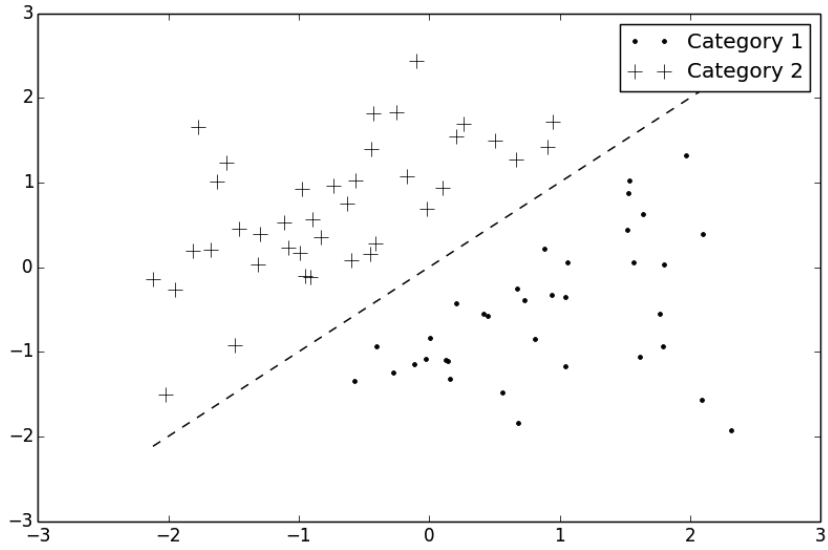


Figure 7: Example of a linearly separable 2-dimensional SVM problem

belong to one of two categories of interest. Then we can write the training data as

$$T = \{(X_i, y_i) \, | i = 1, \ldots, n, X_i \in \mathbb{R}^m, y_i \in \{-1, 1\}\}$$

with each of the n vectors X_i being a sample observation, belonging to a category y_i.

Figure 7 illustrates an example training data set in two dimensions, where the + marks indicate an observation that belongs to one category, and the $\cdot$ marks indicate observations which belong to another category. The objective of an SVM is to fit a model that approximates the dashed line that separates the two categories within the observation space.

That is, SVMs seek to fit a linear separation model to the given training data. This becomes a quadratic optimization problem that takes the form[4]

$$\min_{w,b,\xi} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \xi_i \tag{1}$$

$$\text{s.t. : } y_i \left(w X_i + b\right) \geq 1 - \xi_i \tag{2}$$

$$\xi_i \geq 0 \forall i = 1, \ldots, n \tag{3}$$

where $C > 0$ is a regularization parameter that offers a tunable balance between fidelity to the training data (high values of C) with over-fitting (low

values of C). This problem has the dual formulation

$$\min_a \frac{1}{2} a^T Q a - e^T a$$

$$\text{s.t. : } y^T a = 0$$

$$0 \leq a_i \leq C \forall i = 1, \ldots, n$$

where $Q_{i,j} = X_i^T X_j$ and $e = [1, \ldots, 1]$. Solving this numerically gives classification function $F : \mathbb{R}^m \rightarrow \{-1, 1\}$ of the form

$$F\left(X_{new}\right) = \text{sign}\left(\sum_i a_i y_i X_i^T X_{new}\right)$$

which can assign any new observation X_{new} to one of the two categories presented in the training data, by returning a value of -1 or 1.

Note that both the problem statement (the dual formulation) and the decision function refer to data in the observation space (whether training data or a new observation) only by means of an inner product of the form $X_i^T X_j$. Consider replacing this expression with some transformation function $K\left(X_i, X_{new}\right)$, known as a kernel. The decision function then becomes

$$F\left(X_{new}\right) = \text{sign}\left(\sum_i a_i y_i K\left(X_i, X_{new}\right)\right).$$

This is known in the machine learning community as the "Kernel Trick", and allows for the application of support vector machines to data sets which are not linearly separable within the observation space[4]. For example, consider the data shown in Figure 8, where the two categories of interest are clearly not linearly separable within the observation space.

In this case, one may use a kernel transformation function in order to fit a separation model, such as the one approximated by the dashed line. Table 1 lists a number of popular kernel transformation functions.

Kernel selection is a somewhat nuanced process, but a number of generally applicable best practices have been documented by the machine learning community[11]. For instance, it has been observed that the radial basis kernel performs very well across a wide number of problem types, and does not overfit problems with training data that is linearly separable. As a consequence, many users consider the radial basis function as a robust choice for a default SVM kernel.

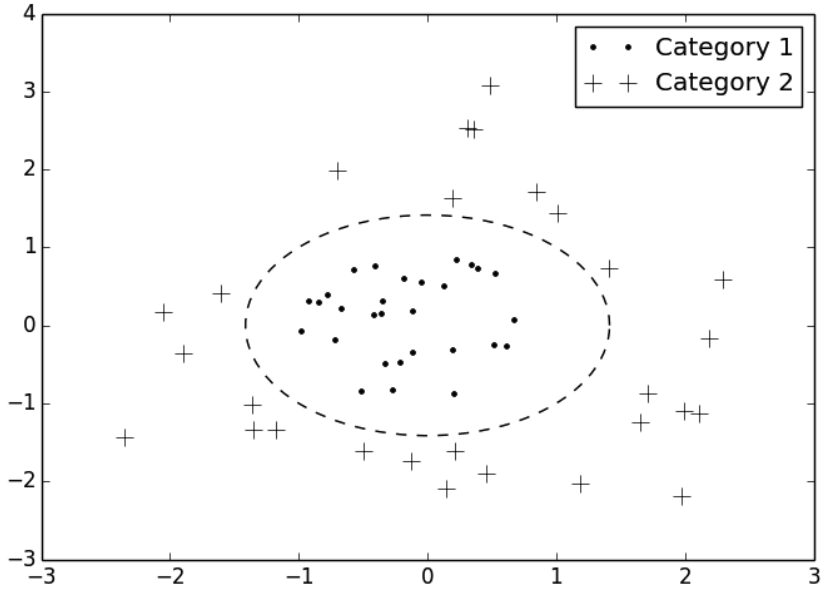


Figure 8: Example of a non-linearly separable 2-dimensional SVM problem

Name	$K(X_i, X_j)$	Parameters
Linear	$X_i^T X_j$	(None)
Polynomial	$\left(X_i^T X_j\right)^p$	$p > 0$
Sigmoid	$\tanh\left(k * X_i^T X_j + d\right)$	$k, d > 0$
Radial Basis Function (RBF)	$\exp\left(-\gamma \|X_i - X_j\|_2^2\right)$	$\gamma > 0$

Table 1: Commonly used SVM kernel functions

Regardless of the choice of kernel function, a value of the tuning parameter C must be also determined by the user. And as seen in Table 1, a number of kernel functions have their own free parameters that must be selected by the user. Strategies for determining free parameters are discussed in Section 4.4.

SVMs can be used in Python code very easily, using the scikit-learn library. Listing 18 illustrates how to construct a very simple SVM in Python. In this example, the SVM instantiated with an RBF kernel and specified parameter values, and is then trained to emulate the XOR operator. The trained SVM is then applied to 3 new inputs (the last of which is "noisy", and not even integer valued), and the SVM is shown to correctly classify them.

Listing 18: Using SVMs In Python

```
from sklearn import svm

# Training data
observations = [[0, 0], [0, 1], [1, 0], [1, 1]]
labels = [0, 1, 1, 0]

# Make an SVM with RBF kernel, and set parameter values
classifier = svm.SVC(kernel='rbf', C=1, gamma=1)

# Fit the SVM to our data
classifier.fit(observations, labels)

# Classify some new data points
print classifier.predict([0,1])
print classifier.predict([0,0])
print classifier.predict([0.9,0.01])
```

4.3 k-means Clustering

k-means clustering is an algorithm that seeks to partition a given set of observations into k distinct sets (or "clusters"), such that each observation belongs to the set that has a mean value closest to the observation. Its purpose is not to provide a classification for future data, but instead to provide unknown insight into a given data set. The data provided does not have to be labeled or pre-classified in any way. However, the user must select a priori the number of clusters to partition the given data into.

Computationally, this problem has the form[1]

$$\min \sum_{i=1}^{k} \sum_{n \in S_i} \|X_i - \mu_i\|^2$$

where each X_i is an observation, S_i represents the set of points in cluster i, μ_i is the mean of cluster S_i, and the minimization is taken over the placement of the observations into each S_i.

There are a number of solution methods that have been implemented to solve this problem efficiently, with the most common being a sequential alternating re-estimation procedure known as Lloyd's Algorithm (for details, see [18]).

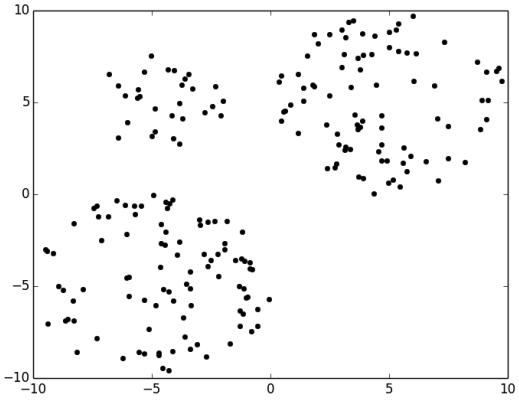
Listing 19 shows a smaller code example of how to use the $k-$means algorithm in Python, using the scikit-learn library. The input data was roughly made to lie in either the first or third quadrants, and the $k-$means algorithm was run with $k = 2$.

Listing 19: Using $k-$means In Python

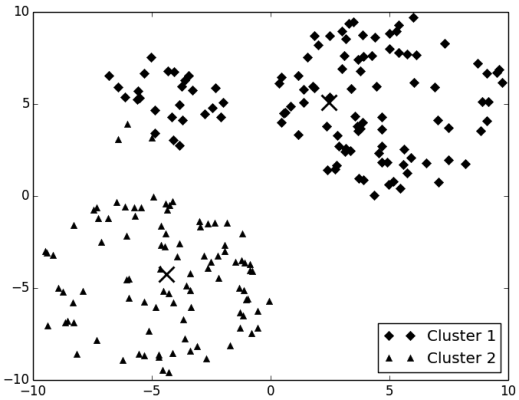
```
from sklearn import cluster

# Some data
data = [[1, 1],[2,1], [1,3], [-1,-1], [-2,-3], [-2,-1]]

# Create instance of k-means, with k=2
kmeans = cluster.KMeans(n_clusters=2)

# Run the algorithm
kmeans.fit(data)

# Gather the results
labels = kmeans.labels_
centroids = kmeans.cluster_centers_

print labels
print
print centroids
```

4.4 Parameter Selection Via Cross-Validation

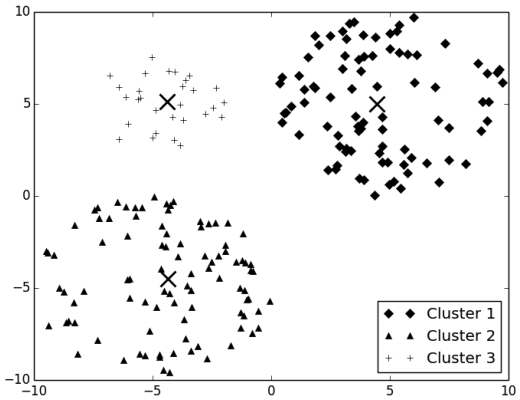
Machine learning methods are rarely block-box methods, in the sense of no configuration being required by a user. Instead, there are often free parameters that must be selected a priori. These parameters often serve the purpose of regularization. That is, they tune a balance between fidelity to the provided data and tractability of the resulting classifier. This is comparable



(a) Original data



(b) $k = 2$



(c) $k = 3$

Figure 9: Graphical example of the $k-$means algorithm. The top image shows the original data, the bottom-left figure shows the clustering of this data with $k = 2$, and the bottom-right shows the clustering with $k = 3$. The large Xs in the bottom two figures show the centroid of each computed cluster.

to the balance that must be maintained when fitting any sort of numerical model.

For example, consider the simpler problem of fitting a polynomial to a collection of data points, as seen in Figure 10. In this case, the degree of the polynomial to be fitted is a free parameter which must be selected. When the degree is one (linear fit), the resulting fit is not very accurate on a point-by-point basis, but generalizes very well outside of the range given in the input data. So it is not very suitable for interpolation within the range of the given data points, but is suitable for extrapolation. When the degree is very high, the fitted model will be able to reproduce all of the input data perfectly, but generalizes extremely poorly. This makes it suitable for continuous interpolation within the range of the original data, but fairly useless for prediction. Thus, it is important to select this parameter so that the resulting model has the correct sought behavior.

The free parameters in any machine learning method together play the exact same role as the polynomial degree in this example, and should be thought of in the same way. For example, the C parameter for Support Vector Machines directly controls the trade-off between fidelity to the training data, and the size of the fitting coefficients that define the resulting SVM model, as seen in Equation 1. The parameters of the kernel functions listed in Table 1 play similar roles.

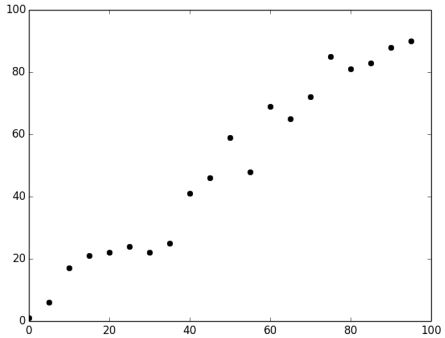
If good values for these parameters are not known from past experience, there are a number of strategies for determining suitable values. The most commonly used procedure is known as cross-validation[23, 16, 11].

In a cross-validation procedure, the data which is available to be used for training is partitioned into two groups: One group is used for training the classifier, the other group is used to validate the resulting classifier after training. This validation is performed by computing the percent of the data points within the validation set which are correctly classified. This procedure is repeated many times using the same parameter values, but each time under a different random partitioning of the available data. A total accuracy measure is computed by averaging over the accuracy computed for each iteration. If the training data that is available is suitably robust for the problem at hand, then this procedure provides a good qualitative estimate for both the accuracy and generalizability of a classification model trained using the given set of parameter values. To determine the best values for a set of parameters, one can then perform numerical optimization or Monte Carlo sampling on this cross-validation procedure.

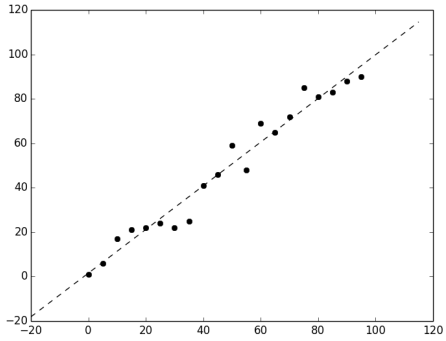
Listing 20 shows an example implementation of a cross-validation procedure in Python. In this example, the free parameters of SVM with a RBF kernel are optimized via Monte Carlo sampling, to determine parameters that produce the most robust classifier.

Listing 20: Cross-validation Procedure In Python

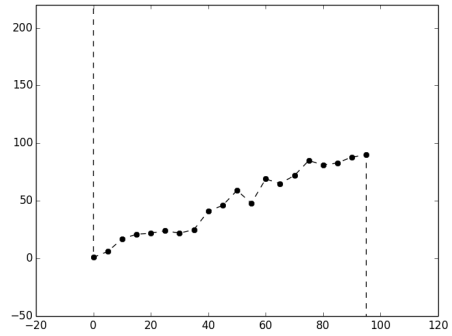
```
from sklearn import svm
from sklearn import cross_validation
import random

def cv_score(observations, labels, C, gamma):
    """
    Cross-validation score for particular C and gamma values
    """
    classifier = svm.SVC(kernel='rbf', C=C, gamma=gamma)
    scores= cross_validation.cross_val_score(classifier,
        observations,
        labels,
        cv=5,
        score_func=metrics.f1_score)
    return np.mean(scores)

def cross_validate(observations, labels, points=100):
    """
    Finds C and gamma values with best Cross-validation score.
    Uses Monte Carlo sampling.
    """
    best = [0, 1, 1]
    for i in xrange(points):
        C = random.randrange(0.1, 10000)
        gamma = random.randrange(0.1, 10000)
        score = cv_score(observations, labels, C, gamma)
        if score > best[0] or (score == best[0] and (C < best[1])):
            best = [score, C, gamma]
            print "New best:", best

observations = [[0, 0], [0, 1], [1, 0], [1, 1]]
labels = [0, 1, 1, 0]

print cv_score(observations, labels, 1, 1)
```



(a) Original data



(b) Linear fit



(c) High-order fit

Figure 10: Graphical example of the balance that must be maintained when choosing a free parameter value when fitting a numerical model. Here, the free parameter is the degree of the polynomial used to fit the given data, shown in (a).

5 Processing Software Organization

This section discusses organization strategies for software written to process single channel modalities, multi-channel modalities, and multi-modal data processing systems. In these systems, each modality may have a variety of sequential processing algorithms that are to be applied to the data, with a classification system as the final step.

Software systems of this type benefit greatly from an object-oriented structure. Object-oriented programming (OOP) is a paradigm which allows for the specification of programmatic constructs, which describe data and functions which operate on that data in a modular and re-usable fashion.

Examples of object-oriented design have already been given in Section 3, where software interfaces to the ISS Imagent, Emotiv Epoc, and Neulog sensors were shown written as Python classes, with a common interface across the three device types. In this way, all of the code needed to read from these devices (whether via serial port, or low level access to a compiled library SDK) and all ancillary data involved (ie. bit maps that specify the communication protocols) can be written, tested, and maintained separately from any application which needs to use these interfaces. Object oriented interfaces also allow for immediate support for interfacing with multiple devices of the same type.

For example, if one wished to write code for two separate Neulog sensor blocks, the code would look like what is shown in Listing 21. A non-object oriented implementation may not scale in as modular a fashion as shown here.

Listing 21: Example Of Using Multiple Class Instances

```
from Neulog import Neulog

device_1 = Neulog("COM3", 9600)
device_2 = Neulog("COM5", 14400)

device_1.scan()
device_2.scan()

while True:
    data1 = device_1.read()
    data2 = device_2.read()
    ...
```

Now, consider all of the data acquisition, data processing, and classification algorithms that have been discussed to this point as separate objects - things

that it would be desirable to implement, test, and maintain in isolation from the other similar parts of a software system. In this way, one can organize an entire data processing software system in a modular way.

5.1 Single Channel Processing

A single modality, single channel device is the simplest such system that we may consider. This kind of processing may be thought of as a sequence of functions being computed on data, with the output of one computation passing as input to the next function in the sequence. Figure 11 shows an example block chart of the data flow involved in a single channel real-time EEG processing and classification system. This particular system operates by collecting a contiguous block of data from a single channel of an EEG device during use by a subject, then computes an index of task engagement e, given as[28]:

$$e = \frac{||\beta||^2}{||\alpha||^2 + ||\theta||^2}$$

where $||.||^2$ denotes spectral power of the specified brainwave type (see Section 3.1.1 for details). The engagement index is then provided to a pre-trained classification algorithm to predict the current state (engaged or un-engaged) of the subject. Listing 22 shows an implementation of this processing in Python. Training the classifier used in this example involves computing engagement indices on collections of prior data, and manually ascribing one of the two cognitive states to those indices. Note that there is nothing which limits this particular system to classification between two states only. If data is available which is believed to delimit between three or more states, then the classifier can be trained to classify data into these states simply by labeling the training data. In production code, this class should also include methods for optimal selection of classifier parameters via cross-validation, as discussed in Section 4.4. However for the sake of brevity, these are omitted in example code shown in this section and the sections which follow.

If it is not clear a priori how many separable states are reflected within a set of training data (or if it is not clear how the data should be labeled), then this is where a clustering algorithm (such as $k-$means) is useful.

Listing 22: Single-Channel EEG Processing Example In Python

```
import numpy as np
```

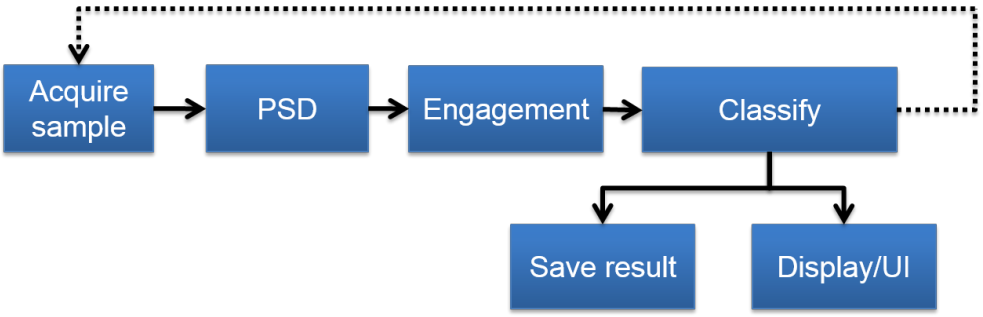


Figure 11: Example data flow for a single-channel EEG classification system.

```
from emotiv import Epoc
from sklearn import svm

class EEGChannel(object):

    def __init__(self, sample_rate=128.):
        """
        Initializes object, sets defaults
        """
        self.sample_rate = sample_rate
        self.raw = np.zeros(10)
        self.psd = np.zeros(10)
        self.freqs = np.zeros(10)
        self.power = [1,1,1]
        self.engagement = 0

        self.classifier = svm.SVC(kernel='rbf', C=1, gamma=1)
        self.state = 0

    def calc_psd(self):
        """
        Computes power spectrum
        """
        N = data.shape[1]
        windowed = np.hamming(N)*data
        self.freqs = float(sample_rate)/N*np.arange(N/2 + 1)
        self.psd = np.abs(np.fft.rfft(windowed, axis = 1))**2

    def calc_band_power(self):
        """
        Computes beta, alpha, and theta levels
        """
        self.power = []
        for band in [[16, 31], [8, 15], [6, 10]]:
            pwr = np.sum(self.psd[:, (self.freqs >= band[0])
                  & (self.freqs <= band[1])], axis=1)
            self.power.append(pwr)

    def calc_engagement(self):
        """
```

```
        Computes theta / (alpha + theta)
        """
        self.engagement = self.power[0] / (self.power[1] + self.power[2])

    def train(self, data, labels):
        """
        Train an SVM classifier
        """
        self.classifier.fit(data, labels)

    def run(self, data):
        """
        Process raw data
        """
        self.raw = data
        self.calc_psd()
        self.calc_band_power()
        self.calc_engagement()

    def classify(self):
        """
        Classify based on current engagement index
        """
        self.run()
        self.state = self.classifier.predict(self.engagement)

# initialize acquistion and processing objects
device = Epoc()
processing = EEGChannel()

# load data and train classifier
train_data = np.loadtxt("eeg_engagement.dat")
train_labels = np.loadtxt("eeg_engagement_labels.dat")
processing.train(train_data, train_labels)

# select EEG channel to use
chan = 7

# initialize data array
data = device.read()[chan].tolist()
buffer_size = 1024
while True:

    new = device.read()[chan]
    data.extend(new)

    if len(data) > buffer_size:
        data = data[-buffer_size:]

    processing.classify(data)
    print processing.state
```

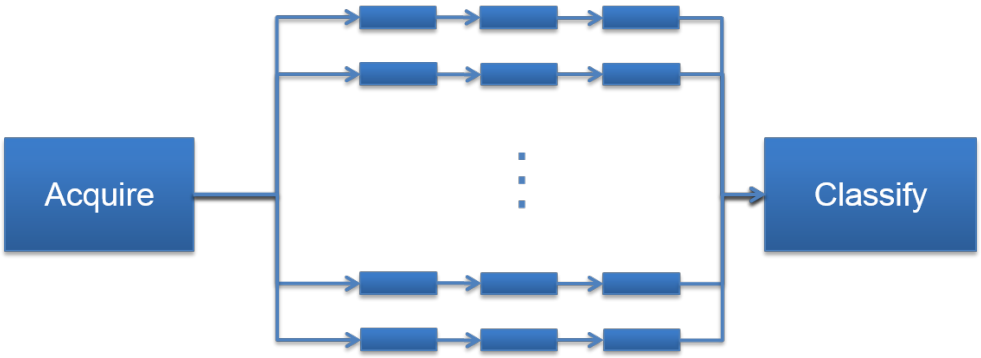


Figure 12: Conceptual multi-channel processing system, with no pre-processing or cross-channel calculations.

5.2 Multiple Channel Processing

A multiple channel (such as fNIRS or EEG), single modality system are considered next. If the processing which is to be performed on each channel is independent of the data for other channels yet identically defined, than the simplest way to implement such a processing system is create instances of single-channel processing objects and make use of their interfaces. If there is not pre-processing or post-processing to be performed which combines inter-channel data in some way, than this structure is sufficient to define the entire processing model. Figure 12 shows a block chart of the data flow involved in a multi channel processing and classification system, which has no pre-processing, post-processing, or cross-channel processing.

In contrast, Figure 13 shows a block chart of the data flow involved in a multi channel processing and classification system, that includes pre-processing, post-processing, and cross-channel processing.

Listing 23 shows an example implementation of a multi-channel EEG processing and classification code which re-uses the object defined for single-channel processing to simply its own definition. This code also pre-computes a cross-correlation matrix and PCA transform between all of the input channels, and implements a post-processing SVM for classification. Each channel is processing to produce the same engagement index described in Section 5.1, but instead of training an SVM for each channel, a single SVM is created which takes the engagement indices from each channel as input.

Listing 23: Multi-Channel EEG Processing Example In Python

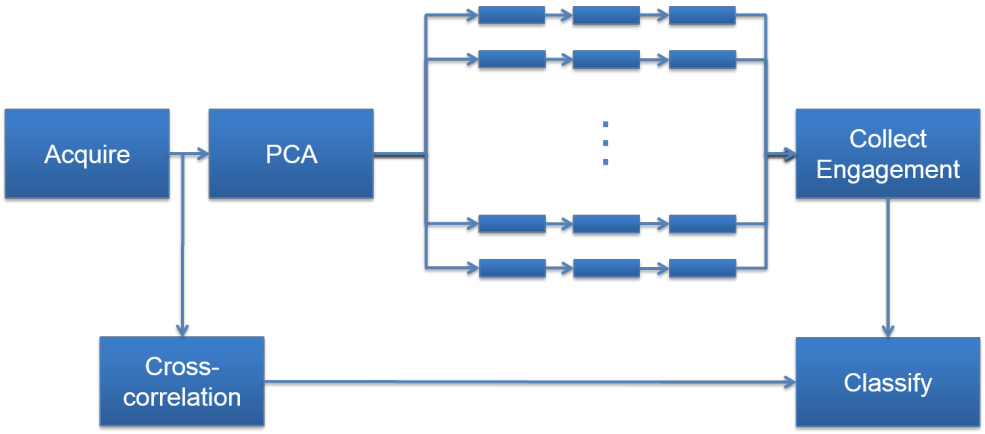


Figure 13: Conceptual multi-channel EEG processing system, with PCA pre-processing and cross-correlation calculations.

```
import numpy as np
from emotiv import Epoc
from sklearn import svm
from sklearn.decomposition import PCA
from EEG import EEGChannel

class EEGMultiChannel(object):

    def __init__(self, sample_rate=128.):
        """
        Initializes object, sets defaults
        """
        self.channels = [EEGChannel(sample_rate) for i in xrange(14)]

        self.classifier = svm.SVC(kernel='rbf', C=1, gamma=1)
        self.state = 0

        self.PCA = PCA()
        self.correlation_matrix = np.zeros((14, 14))

    def train(self, data, labels):
        """
        Train a PCA transform pre-processor and
        an SVM classifier
        """
        self.classifier.fit(data, labels)
        self.PCA.fit(data)

    def classify(self):
        """
        Classify based on engagement indices of each channel.
        """
        self.run()
        self.state = self.classifier.predict(self.engagement_values)
```

```
    def run(self, raw_data):
        """
        Computes correlation matrix of channels, then
        process them using a pre-fitted PCA.
        Engagement indices for each channel are then computed
        """
        self.correlation_matrix = numpy.corrcoef(raw_data)

        data = PCA.transform(raw_data)

        self.engagement_values = []
        for i, channel in enumerate(self.channels):
            channel.raw = data[i]
            channel.run()
            self.engagement_values.append(channel.engagement)

# initialize acquistion and processing objects
device = Epoc()
processing = EEGMultiChannel()

# load data and train classifier
train_data = np.loadtxt("eeg_engagement.dat")
train_labels = np.loadtxt("eeg_engagement_labels.dat")
processing.train(train_data, train_labels)

# initialize data array
data = device.read()
buffer_size = 1024
while True:

    new = device.read()
    data = np.concatenate((data, new), axis  = 1)

    if data.shape[1] > buffer_size:
        data = data[:,-buffer_size:]

    processing.classify(data)
    print processing.state
```

5.3 Multi-modal Processing

Finally, the design of a multi-modality system (where each modality may be single or multi-channel) is considered. Figure 14 shows a block chart of the data flow involved in a multi-modal processing and classification system, involving a layer of cross modality processing and classification. For example, consider a system which involves real-time processing of both fNIRS and EEG data, from an ISS Imagent and Emotiv Epoc (respectively). In this system, each modality will be processed and classified using their own defined multi-channel processing objects. These objects will use data processed for each

channel as input to an SVM classifier. The classification results from both of these modalities will then be used as input to a second classification layer, which will use Naive Bayes as a classification algorithm. This effectively implements a voting scheme between the different modalities, to determine a single combined estimate of operator state.

The described two-modality data processing and classification can be implemented by writing an object which creates an instance of both the EEG processing and fNIRS processing class, and passes relevant data to each of these instances appropriately.

However, on the data acquisition side, one issue that may come to mind is that these two devices operate at very different sample rates. The Epoc operates at 128 Hz, and the Imagent at 6 Hz. If we were sequentially acquiring data from both of these devices within a single loop of a software program, this program would only be able to respond and produce an updated classification result (at the end of the main loop) at a rate no faster than the slowest device, which in this case would be the fNIRS instrument. EEG data may be lost, if the length of time between EEG acquisitions is long enough that the device buffer fills. So as additional modalities are added, a large disparity in sample rates may lead to a system which cannot operate for effectively than the sum of its individual parts, if the acquisition is handled this simple sequential way.

One way around this limitation is to acquire data asynchronously from each instrument using two independent program scripts, which will update a real-time database system concurrently. The main program which will analyze and classify this data will read from this database at a much faster rate than the acquisition from either of the two devices, and can update it's analysis and classification measures continuously.

Listings 24 and 25 show two programs that acquire data from an Imagent and Epoc and write and update the data to two separate keys within the same real-time database. These programs are started and run as background processes, at the same time as the main program shown in Listing 26. The real-time database object shown has many possible implementations, using fast data storage implementations such as MongoDb[13] or Redis[14], together with their respective Python interfaces. This database object effectively provides a scalable fast shared memory between multiple running programs, even

[13]https://www.mongodb.org/

[14]http://redis.io/

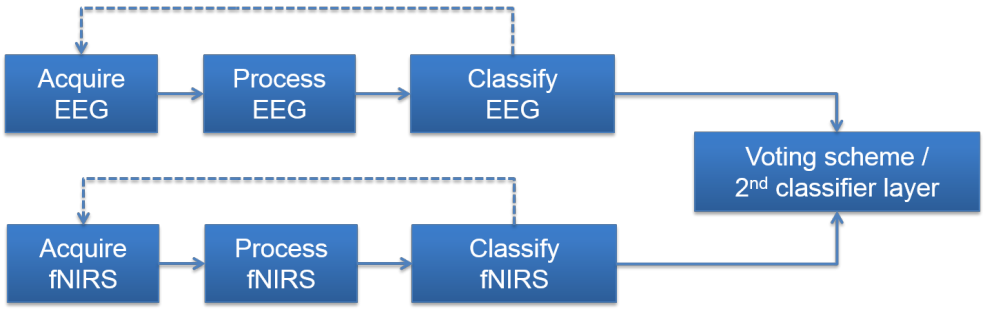


Figure 14: Conceptual multi-modal processing system

between programs running on different computers but within the same local network. This strategy will allow for flexible implementation of a multi-modal classification system, that can scale to meet needs of additional modalities which may not yet be identified.

Listing 24: ISS Imagent Acquistion And Database Writing

```
from ISS import Imagent
import database

# initialize acquistion and processing objects
db = database(ip="localhost", port=27017)
device = Imagent('COM5', 115200)

# initialize data array
data = device.read()
buffer_size = 48
while True:

    new = device.read()
    data = np.concatenate((data, new), axis  = 1)

    if data.shape[1] > buffer_size:
        data = data[:,-buffer_size:]

    # update data array in the database
    db.set("fnirs", data)
```

Listing 25: Emotiv Epoc Acquistion And Database Writing

```
from emotiv import Epoc
import database

# initialize acquistion and processing objects
db = database(ip="localhost", port=27017)
device = Epoc()

# initialize data array
data = device.read()
buffer_size = 1024
while True:

    new = device.read()
    data = np.concatenate((data, new), axis  = 1)

    if data.shape[1] > buffer_size:
        data = data[:,-buffer_size:]

    # update data array in the database
    db.set("eeg", data)
```

Listing 26: Multi-Modal Processing System Implementation

```
import numpy as np
import database
from sklearn.naive_bayes import GaussianNB
from EEG_Mult import EEGMultiChannel
from fNIRS_mult import fNIRSMultiChannel

class MultiModal(object):

    def __init__(self):
        self.eeg = EEGMultiChannel()
        self.fnirs = fNIRSMultiChannel()
        self.classifier = GaussianNB()

    def train(self, data, labels):
        """
        Trains a second-level classification system

        data (2-dim list) : predicated states from EEG and fNIRS svms
        labels : true states
        """

        self.classifier.fit(data, labels)

    def classify(self. eeg_data, fnirs_data):
        self.eeg.classify(eeg_data)
        self.fnirs.classify(fnirs_data)

        estate = self.eeg.state
        fstate = self.fnirs.state

        self.state = self.classifier.predict([estate, fstate])

# initialize acquistion and processing objects
db = database(ip="localhost", port=27017)
processing = MultiModal()

# load data and train classifier
eeg_data = np.loadtxt("eeg_engagement.dat")
eeg_labels = np.loadtxt("eeg_engagement_labels.dat")
processing.eeg.train(eeg_data, eeg_labels)

fnirs_data = np.loadtxt("fnirs_engagement.dat")
fnirs_labels = np.loadtxt("fnirs_engagement_labels.dat")
processing.fnirs.train(fnirs_data, fnirs_labels)

mm_data = np.loadtxt("mm_engagement.dat")
mm_labels = np.loadtxt("mm_engagement_labels.dat")
processing.train(mm_data, mm_labels)

while True:
    eeg_data = db.get("eeg")
    fnirs_data = db.get("fnirs")

    processing.classify(eeg_data, fnirs_data)
    print processing.state
```

References

[1] Bishop, Christopher M. "Neural networks for pattern recognition". Oxford university press, 1995.

[2] Chang, Chih-Chung, and Chih-Jen Lin. "LIBSVM: a library for support vector machines." ACM Transactions on Intelligent Systems and Technology (TIST) 2.3 (2011): 27.

[3] Cooley, J, et. al., The application of the fast Fourier transform algorithm to the computation of spectra and cross-spectra, J. Sound Vib., 12 (1970), pp. 339–352.

[4] Cortes, Corinna and Vapnik, V. "Support-vector networks." Machine learning 20.3 (1995): 273-297.

[5] Delpy, D. T., et al. "Estimation of optical pathlength through tissue from direct time of flight measurement." Physics in medicine and biology 33.12 (1988): 1433.

[6] Fatma Guler, N., and Elif Derya Ubeyli. "Multiclass support vector machines for EEG-signals classification." Information Technology in Biomedicine, IEEE Transactions on 11.2 (2007): 117-126.

[7] Goodman, Steven N. "Toward evidence-based medical statistics. 1: The P value fallacy." Annals of internal medicine 130.12 (1999): 995-1004.

[8] Gratton, E., et al. "The possibility of a near-infrared optical imaging system using frequency-domain methods." Proc. III Int. Conf. Peace through Mind/Brain Sci. Vol. 183. 1990.

[9] Gratton, Enrico, et al. "Measurement of brain activity by near-infrared light." Journal of Biomedical Optics 10.1 (2005): 011008-01100813.

[10] Harrivel, Angela R., et al. "Monitoring attentional state with fNIRS." Frontiers in human neuroscience 7 (2013).

[11] Hsu, Chih-Wei, Chih-Chung Chang, and Chih-Jen Lin. "A practical guide to support vector classification." (2003).

[12] Huppert, Theodore J., et al. "Quantitative spatial comparison of diffuse optical imaging with blood oxygen level-dependent and arterial spin labeling-based functional magnetic resonance imaging." Journal of biomedical optics 11.6 (2006): 064018-064018.

[13] Hyvärinen, A. Independent Component Analysis, Adaptive and Learning Systems for Signal Processing, Communications and Control Series, Wiley-Blackwell, 2001.

[14] Joachims, T. (1998). "Text categorization with support vector machines: Learning with many relevant features" (pp. 137-142). Springer Berlin Heidelberg.

[15] Kocsis, L., P. Herman, and A. Eke. "The modified BeerLambert law revisited." Physics in medicine and biology 51.5 (2006): N91.

[16] Kohavi, Ron. "A study of cross-validation and bootstrap for accuracy estimation and model selection." IJCAI. Vol. 14. No. 2. 1995.

[17] Liu, Hanli, et al. "Determination of optical properties and blood oxygenation in tissue using continuous NIR light." Physics in medicine and biology 40.11 (1995): 1983.

[18] Lloyd, Stuart. "Least squares quantization in PCM." Information Theory, IEEE Transactions on 28.2 (1982): 129-137.

[19] Lotte, Fabien, et al. "A review of classification algorithms for EEG-based braincomputer interfaces." Journal of neural engineering 4 (2007).

[20] Matthews, Fiachra, et al. "Hemodynamics for brain-computer interfaces." Signal Processing Magazine, IEEE 25.1 (2008): 87-94.

[21] S. Mallat, A Wavelet Tour of Signal Processing, Academic Press, 2008.

[22] McCallum, Andrew, and Kamal Nigam. "A comparison of event models for naive bayes text classification." AAAI-98 workshop on learning for text categorization. Vol. 752. 1998.

[23] Michie, Donald, David J. Spiegelhalter, and Charles C. Taylor. "Machine learning, neural and statistical classification." (1994).

[24] Mitchell, Tom M., et al. "Classifying instantaneous cognitive states from fMRI data." American medical informatics association annual symposium. 2003.

[25] Mouro-Miranda, Janaina, et al. "Classifying brain states and determining the discriminating activation patterns: support vector machine on functional MRI data." Neuroimage 28.4 (2005): 980-995.

[26] Oliphant, Travis. "Python for scientific computing". Computing in Science & Engineering 9.3 (2007): 10-20.

[27] Osuna, Edgar, Robert Freund, and Federico Girosi. "Training support vector machines: an application to face detection." Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on. IEEE, 1997.

[28] Pope, Alan T., Edward H. Bogart, and Debbie S. Bartolome. "Biocybernetic system evaluates indices of operator engagement in automated task." Biological psychology 40.1 (1995): 187-195.

[29] Schneider, Karl-Michael. "A comparison of event models for Naive Bayes anti-spam e-mail filtering." Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics-Volume 1. Association for Computational Linguistics, 2003.

[30] Schnitzler, Alfons, and Joachim Gross. "Normal and pathological oscillatory communication in the brain." Nature reviews neuroscience 6.4 (2005): 285-296.

[31] Villringer, Arno, and Britton Chance. "Non-invasive optical spectroscopy and imaging of human brain function." Trends in neurosciences 20.10 (1997): 435-442.

[32] Xu, Rui, and Donald Wunsch. "Survey of clustering algorithms." Neural Networks, IEEE Transactions on 16.3 (2005): 645-678.

[33] Zhang, Harry. "The optimality of naive Bayes." A A 1.2 (2004): 3.