AI Assessed Exercise

Ross Meikleham 1107023m

December 11, 2015

1 Chapter 1 - Design

- **Performance** A metric used to tell how "well" the intelligent agent is performing, it can also be described as a criteria for success.
- **Environment** An environment can be through of as a state in which the agent can percieve data from as well as changing its state by acting upon it. An environment may be static, or change over time (dynamic), An environment can contain other agents (Multiagent environment). An environment also limits the circumstances the agent can deal with.
- **Sensors** How the intelligent agent percieves its environment
- **Actuators** Actuators let the intelligent agent act on its environment.

In the laboratory experiment the PEAS terms are:

- **Performance** Average loss when predicting the class the data belongs to must be below a certain percentage.
- **Environment** Audio samples
- **Sensors** Reading short term energy and magnitude signals, as well as zero crossing rate from the audio data.
- **Actuators** Generates discriminant functions to be able to predict whether the audio contains voice, or is silent.

2 Chapter 2 - Theory

2.1 Extracting Short Term Signals

The properties of a signal are stable in the short-term but change, so we need to use a sliding window that spans the entire signal at regular intervals of time. In this exercise we use a rectangular window. The samples inside a rectangular window have the value 1, and samples outside the window have the value 0.

A rectangular window is defined as follows:

$$w[n] = \begin{cases} 1, & 0 \leq n \leq N-1 \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

2.1.1 Extracting Energy

The equation for calcuating the energy is as follows:

$$E[n] = \sum_{k=-\infty}^{\infty} \frac{(s[k])^2}{N}.w[n-k] \tag{2}$$

However, since we're using a rectangular window; we know that $w[n]$ is 0 for k values in which $n-k$ is outside of the window, and as a result the equation at these k values will always result in 0. So we can restrict the sum over the values in which $n-k$ is inside the window.

From this we can extract the energy with the rectangular window using the following equation:

$$E[n] = \sum_{k=n-N+1}^{n} \frac{(s[k])^2}{N}.w[n-k] \tag{3}$$

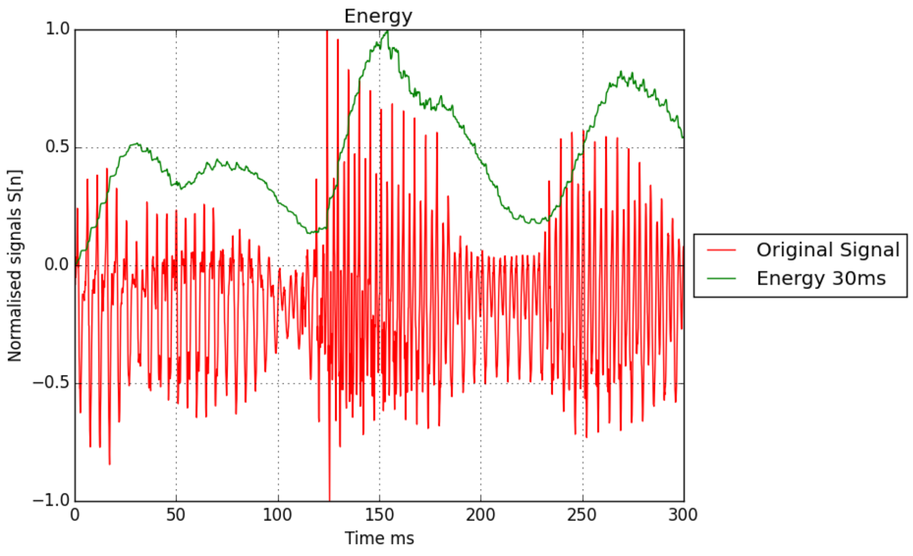


Figure 1: Plot of Energy on a 300ms sample

2.1.2 Extracting Magnitude

The equation for calculating the magnitude is as follows:

$$E[n] = \sum_{k=-\infty}^{\infty} \frac{|s[k]|}{N}.w[n-k] \tag{4}$$

Using the same reasoning as in section 2.1.1 we can extract the magnitude with the rectangular window using the following equation:

$$E[n] = \sum_{k=n-N+1}^{n} \frac{|s[k]|}{N}.w[n-k] \tag{5}$$

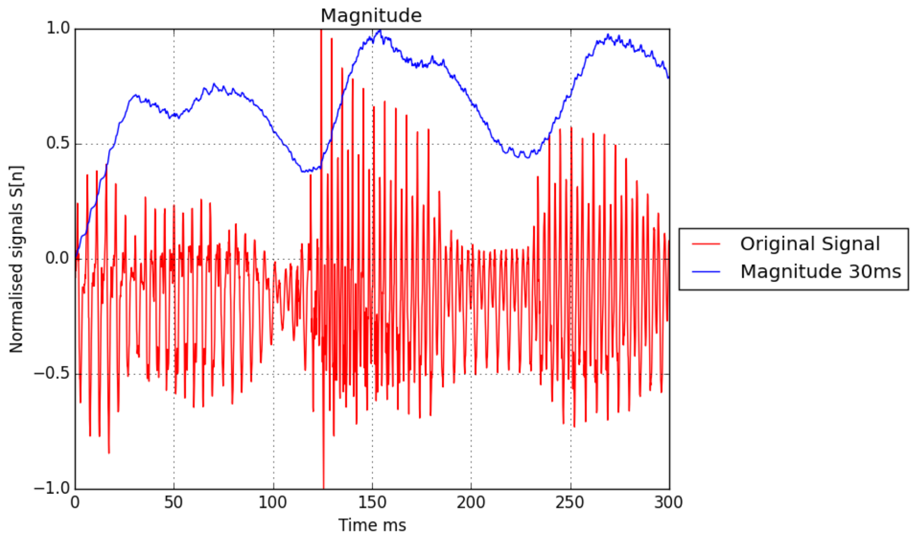


Figure 2: Plot of Magnitude on a 300ms sample

2.1.3 Extracting Zero Crossing Rate

The Zero Crossing Rate provides the rate at which a signal changes from positive to negative, or vice-versa.

The equation for calculating the Zero-Crossing Rate(ZCR) is as follows:

$$E[n] = \frac{1}{2N} \sum_{k=-\infty}^{\infty} |sign(s[k] - sign(s[k-1]])|.w[n-k] \tag{6}$$

where

$$sign[n] = \begin{cases} -1, & n < 0 \\ 1, & n > 0 \\ 0, & n = 0 \end{cases} \tag{7}$$

Using the same reasoning as in section 2.1.1 we can extract the ZCR with the rectangular window using the following equation:

$$E[n] = \frac{1}{2N} \sum_{k=n-K+1}^{N} |sign(s[k] - sign(s[k-1]])|.w[n-k] \tag{8}$$

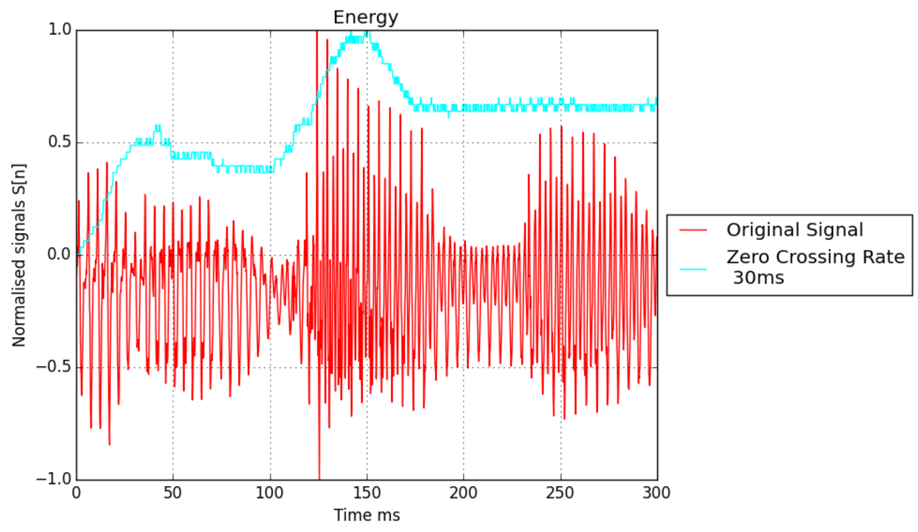


Figure 3: Plot of ZCR on a 300ms sample

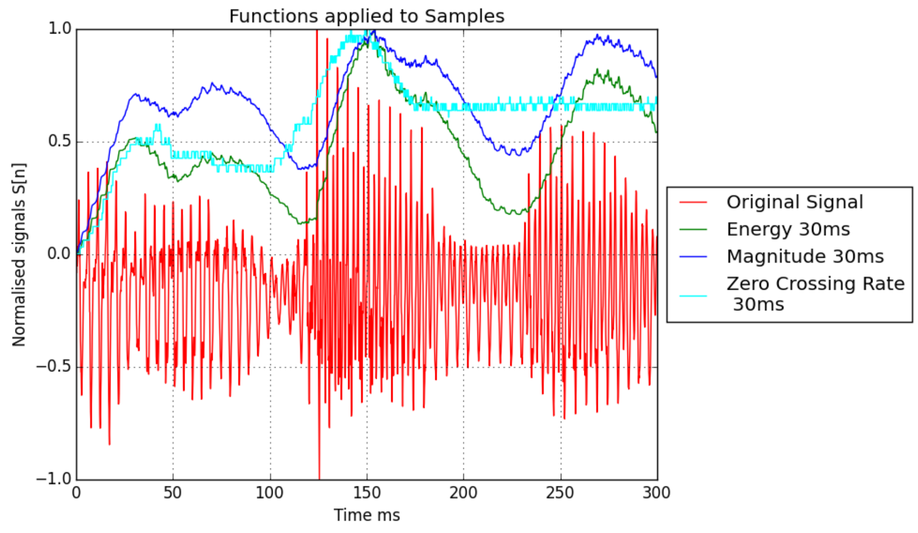


Figure 4: Energy, Magnitude, ZCR, all plotted at once

2.2 Decisions

The exercise is a binary classification problem in that we need to determine if a given sample contains speech or not. The classification property is the presence of speech.

$R(B_i, \vec{x})$ is called Bayes Risk, where in this case

$$R(B_1|\vec{x}) = \pi(B_1|C_1)p(C_1|\vec{x}) + \pi(B_1|C_2)p(C_2|\vec{x}) \tag{9}$$

$$R(B_2|\vec{x}) = \pi(B_2|C_1)p(C_1|\vec{x}) + \pi(B_2|C_2)p(C_2|\vec{x}) \tag{10}$$

where B_i is the correct action for decision C_i. $\vec{x}$ is a set of features.

We can manipulate the Risk equations above to get the following:

$$\frac{p(\vec{x}|C_1)}{p(\vec{x}|C_2)} > \frac{\pi(B_1|C_2) - \pi(B_2|C_2)}{\pi(B_2|C_1) - \pi(B_1|C_1)} . \frac{p(C_2)}{p(C_1)} \tag{11}$$

If the likelihood ratio is larger than the expression on the right hand side, the decision made is B_1 otherwise the decision made is B_2

We need to be able to calculate a value for the amount of loss calculated from the prediction value compared to the actual value.

For this exercise we use a Zero-One Loss function to calculate the loss, which is defined as follows:

$$\pi(B_i|C_i) = \begin{cases} 0, & i == j \\ 1, & i \neq j \end{cases} \tag{12}$$

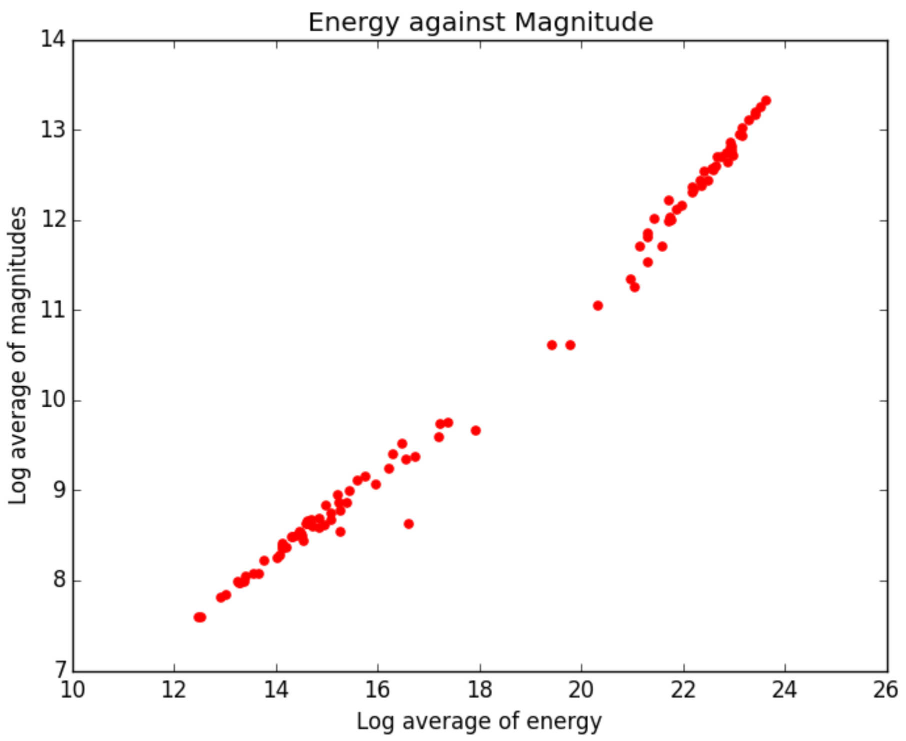
2.3 Feature Values

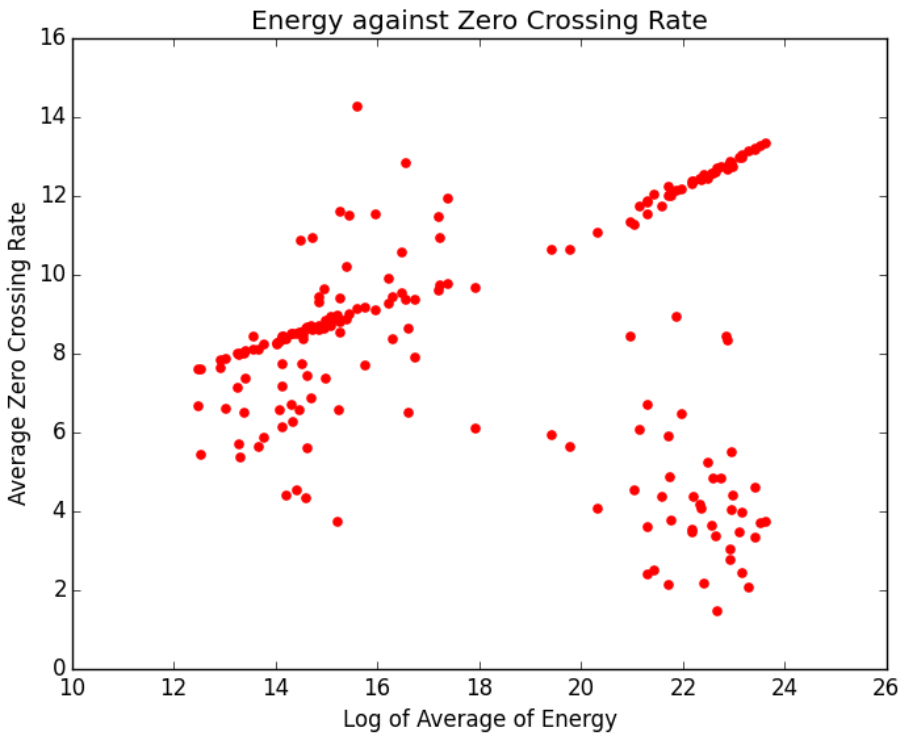
We have three feature values for each sample which we use to predict which class a sample belongs to.

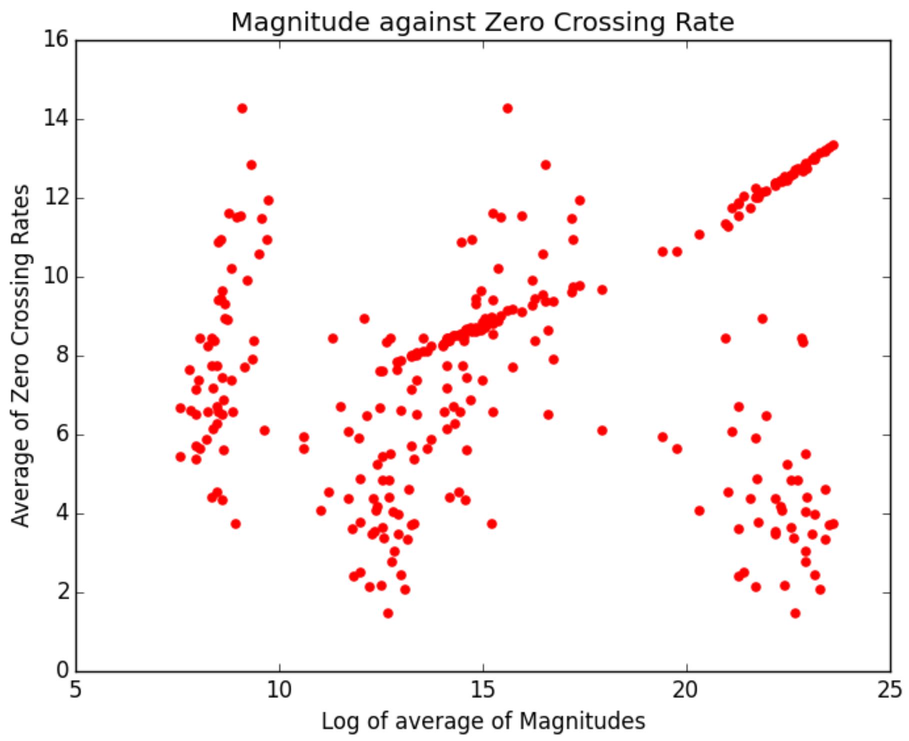
- The logarithm of the average value of the short-term energy signal
- The logarithm of the average value of the short-term magnitude signal
- The average value of the Zero Crossing Rate signal

These values can be computed after using the methods described in section 2.1 to extract the ZCRs, short-term energy signals, and the short-term magnitude signals.

With the given set of 100 samples we then compute the feature values for each sample and then plot them against each other; resulting in the following graphs:


Energy against Magnitude
Log average of magnitudes
Log average of energy
14
13
12
11
10
9
8
7
10
12
14
16
18
20
22
24
26


Energy against Zero Crossing Rate
Average Zero Crossing Rate
Log of Average of Energy
0
2
4
6
8
10
12
14
16
10
12
14
16
18
20
22
24
26



2.4 Gaussian Discriminant

The following is a set of discriminant functions:

$$G = \{\gamma_1(\vec{x}), ..., \gamma_L(\vec{x})\} \tag{13}$$

We select the largest $\gamma_k(\vec{x})$ in G, the decision made is then C_k.

For our case using a Zero-One loss function we obtain the following discriminant functions:

$$\log(\gamma_i(\vec{x})) \simeq \log(p(\vec{x}|C_i)) + \log(p(C_i)) \tag{14}$$

The Univariate Gaussian is a probability density function

$$p(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{15}$$

Where μ is the mean, and σ^2 is the variance.

We assume that the distribution of each of our features is a guassian distribution.

We then get the following Gaussian discriminant functions:

$$\log(\gamma_i(\vec{x})) = -\log\sqrt{2\pi\sigma} - \frac{(x-\mu_i)^2}{2\sigma_i^2} + \log(p(C_i)) \tag{16}$$

There is one problem with using a univariate gaussian discriminant in this case as we have 3 seperate feature variables, and a univariate gaussian only operates on a single variable. We need to modify our discriminant functions to work with this.

$$M_i = \{\alpha_{1_i}(\vec{x}_i), ..., \alpha_{L_i}(\vec{x}_i)\} \tag{17}$$

$$\begin{aligned}\log(\gamma_k(\vec{x}))) &= \log(\alpha_{k_1}(\vec{x})) + \log(\alpha_{k_2}(\vec{x})) + \log(\alpha_{k_3}(\vec{x})) \\ &= \log(\alpha_{k_1}(\vec{x}))\alpha_{k_2}(\vec{x})\alpha_{k_3}(\vec{x}))\end{aligned} \tag{18}$$

Where M_i is a set of univariate gaussian distribution functions on the ith feature, for $1 \leq i \leq 3$.

From this we can generate gaussian discriminant functions from a subset of our data to predict whether a sample has speech in it or not.

3 Chapter 3 - Experiments

From our samples we extract the features using a window size of 15ms, then we randomly split the data into 10 disjoint subsets, ensuring that each subset contains equal amounts of voice samples and silence samples.

We then take one of the subsets as a test set and then concatonate the reminaing subsets into a single training set (using 90% of the data as test data, and 10% as training data). We generate our gaussian discriminant functions on the training data using the method described in section 2.4. We then use the gaussian discriminant functions to predict which class each sample in the training set belongs to and compute the loss using our loss function. We then average the total loss.

We then take another subset as the training set and perform the same steps, and add the average loss to the previous average loss.

We repeat this until all subsets have been used as the training set, and divide the total average loss over the amount of subset to obtain the average.

With this I averaged a loss of 0.05. Which corresponds to 5%. So the cross validation approach was around 95% accurate.

A Lab1 Processing Code

(Haskell)

A.1 Lab1.hs

```haskell
module AI.Lab1.Lab1 where

import Data.Maybe
import System.IO
import qualified Data.Vector.Unboxed as VU

-- | Sampling time for all samples in milliseconds
samplingTime :: Int
samplingTime = 300


-- | Calculate the sampling rate from a signal
samplingRate :: [Int] -> Double
samplingRate samples = fromIntegral (length samples) /
                       (fromIntegral (samplingTime) / 1000.0)


-- Apply ideal operator delay to a signal (S[n]) for
-- a delay of m ms
idealOperatorDelay :: [Int] -> Int -> [Int]

--Delaying by 0 ms gives back the same signal
idealOperatorDelay signal 0 = signal

idealOperatorDelay signal ms =
        -- Remove last n ms from signal
        -- Pad the front n ms with 0
        take n (repeat 0) ++
            take (length signal - n) signal
    where
          samplesPerMs = (length signal `div` samplingTime)
          n = samplesPerMs * ms


-- Apply moving average with k1, k2 ms
-- to a given signal
movingAverage :: [Int] -> Int -> Int -> [Double]
movingAverage signal 0 0 = map fromIntegral signal
movingAverage signal k1 k2 =
    map y [0,1..(length signal - 1)]
    where
          samplesPerMs = (length signal `div` samplingTime)
          nk1 = samplesPerMs * k1
          nk2 = samplesPerMs * k2
          y :: Int -> Double
          y n = let a = max 1 (n - nk1)
                    b = min (length signal - 1) (n + nk2)
                in  fromIntegral (sum $ (take (b - a + 1)) $
                                         drop (a - 1) signal) /
                    fromIntegral (b - a + 1)


-- Rectangular Window function which takes the size of the
-- window "win_sz" and "n", returns 1 if n is inside the window,
-- 0 otherwise
rectWindow :: Int -> Int -> Int
rectWindow win_sz n
    | n < 0 = 0 -- Left of Window
    | n >= win_sz = 0 -- Right of Window
    | otherwise = 1 -- Inside the Window


-- Apply convolution to a signal with a window
-- of given length in milliseconds
convolute :: VU.Vector Int -> Int -> VU.Vector Int
convolute signal win_sz =
            VU.map y $ VU.fromList [0..(numSamples - 1)]

  where y :: Int -> Int
        y n = VU.foldl' (+) 0 $ -- Sum the results
                -- s[k] * w[n - k]
                VU.map (\k -> (signal VU.! k) *
                              (rectWindow win_sz_samples (n - k))) $
                     -- k values
                     VU.fromList [(max (n - win_sz_samples + 1) 0) ..
                                  (min n (numSamples - 1))] -- k values

        -- Number of samples in the Window of win_sz milliseconds
        win_sz_samples = (numSamples `div` samplingTime) * win_sz
        -- Total number of samples supplied to the convolution function
        numSamples = VU.length signal


-- Calculate energy for a given signal with a window
-- of given length in milliseconds
energy :: VU.Vector Int -> Int -> VU.Vector Double
energy signal win_sz = VU.map e $ VU.fromList [0..(numSamples - 1)]

  where e :: Int -> Double
        -- sum(s[k]^2 * w[n-k])/N
        e n = (fromIntegral $ sumRes n) / (fromIntegral win_sz_samples)

        sumRes :: Int -> Int
        sumRes n =
            VU.foldl'(+) 0 $ -- Sum results
                -- s[k]^2 * w[n-k]
                VU.map (\k -> ((signal VU.! k) ^ 2) *
                       (rectWindow win_sz_samples (n - k))) $
                        -- k values
                        VU.fromList [(max (n - win_sz_samples + 1) 0) ..
                                     (min n (numSamples - 1))]

        -- Number of samples in the Window of win_sz milliseconds
        win_sz_samples = (numSamples `div` samplingTime) * win_sz
        -- Total number of samples supplied to the convolution function
        numSamples = VU.length signal


-- Calculate magnitude for a given signal with a window
-- of given length in milliseconds
magnitude :: VU.Vector Int -> Int -> VU.Vector Double
magnitude signal win_sz = VU.map m $ VU.fromList [0..(numSamples - 1)]

  where m :: Int -> Double
        -- sum(|s[k]| * w[n-k])/N
        m n = (fromIntegral $ sumRes n) / (fromIntegral win_sz_samples)

        sumRes n = VU.foldl'(+) 0 $ -- Sum results
                     -- |s[k]| * w[n-k]
                     VU.map (\k -> ((abs (signal VU.! k))) *
                            (rectWindow win_sz_samples (n - k))) $
                         -- k values
                         VU.fromList [(max (n - win_sz_samples + 1) 0) ..
                                      (min n (numSamples - 1))]

        -- Number of samples in the Window of win_sz milliseconds
        win_sz_samples = (numSamples `div` samplingTime) * win_sz
        -- Total number of samples supplied to the convolution function
        numSamples = VU.length signal


-- Calculate zero crossing rate for a given signal with a window
-- of given length in milliseconds
zeroCrossingRate :: VU.Vector Int -> Int -> VU.Vector Double
zeroCrossingRate signal win_sz =
    VU.map m $ VU.fromList [0..(numSamples - 1)]

  where m :: Int -> Double
        -- sum(|s[k]| * w[n-k])/N
        m n = (fromIntegral $ sumRes n) /
                    (2 * (fromIntegral win_sz_samples))

        sumRes n = VU.foldl'(+) 0 $ -- Sum results
```

```haskell
                      -- |sgn(s[k] - sgn(s[k-1])| * w[n-k]
                     VU.map (\k -> (abs (signum (signal VU.! k) -
                                        (signum (signal VU.! (k - 1))))) *
                                        (rectWindow win_sz_samples (n - k)))
                            -- k values
                            $ VU.fromList
                                    [(max (n - win_sz_samples + 1) 1) ..
                                     (min n (numSamples - 1))]

        -- Number of samples in the Window of win_sz milliseconds
        win_sz_samples = (numSamples `div` samplingTime) * win_sz
        -- Total number of samples supplied to the convolution function
        numSamples = VU.length signal


-- Write Samples to CSV file
plotToCsv :: Real a => String -> [a] -> IO ()
plotToCsv name graph =
    writeFile (name ++ ".csv") $
        unlines $ map (\(x, y) -> concat [show x, ",", show y]) $
                      points $ map realToDouble graph
  where

    -- Generate (x,y) points from a signal
    points :: [Double] -> [(Double, Double)]
    points samples = zip
                -- x coodinates
                (map (\x -> fromIntegral x *
                                (fromIntegral samplingTime /
                                (fromIntegral $ length samples))
                     )  [0..])
                 -- y coordinates
                 samples

    realToDouble :: (Real a) => a -> Double
    realToDouble = fromRational . toRational
```

A.2 Main.hs

```haskell
import System.IO
import qualified Data.Vector.Unboxed as VU

import AI.Lab1.Lab1

labFilePath :: String
labFilePath = "laboratory.dat"


main :: IO ()
main = do
```

```haskell
handle <- openFile labFilePath ReadMode
contents <- hGetContents handle

-- Obtain a list of samples (S[n]) as strings from file
let strSamples = lines contents

-- Convert samples to integers
let samples = map (\s -> (read s :: Int)) strSamples

-- Calculate sampling rate
putStr $ "Sampling rate is: " ++
            (show  $ samplingRate samples) ++ "Hz\n"

-- Generate CSV file of original data
plotToCsv "Laboratory" samples

-- Calc ideal operator delays for 5, 10, and 15 ms, and generate CSV
-- files
let iods = map (idealOperatorDelay samples) [5, 10, 15]
mapM_ (uncurry plotToCsv) $ zip ["Delay5", "Delay10", "Delay15"] iods

-- Calc moving averages for k1=k2=5, 10, and 15ms, and generate
-- CSV files
let mas =  map (\(k1,k2) -> movingAverage samples k1 k2)
                [(5, 5), (10, 10), (15, 15)]

mapM_ (uncurry plotToCsv) $
    zip ["MAverage5", "MAverage10", "MAverage15"] mas

-- Calc convolution for window size of 10ms, and generate CSV file
let cvs = convolute (VU.fromList samples) 10
plotToCsv "Convolution" (VU.toList cvs)

-- Calc short term energy signal for 30ms, and generate CSV file
let ste = energy (VU.fromList samples) 30
plotToCsv "Energy" (VU.toList ste)

-- Calc short term magnitude for 30ms, and generate CSV file
let md = magnitude (VU.fromList samples) 30
plotToCsv "Magnitude" (VU.toList md)

-- Calc short term ZCR for 30ms, and generate CSV file
let zcr = zeroCrossingRate (VU.fromList samples) 30
plotToCsv "ZeroCrossingRate" (VU.toList zcr)

hClose handle
```

B Lab1 Plotting Code

(Python)

```
# Plots the sample data generated by the Haskell program
import matplotlib.pyplot as pl
import numpy as np

ax = pl.subplot(111)

# Normalise a list of samples between 1.0 and 0.0
def positiveNormalise(data, minV = None, maxV = None):
    if minV is None:
        minV = min(data)

    if maxV is None:
        maxV = max(data)

    def normaliseItem(item):
        return (item - minV) / (maxV - minV)

    return [normaliseItem(i) for i in data]

# Normalise a list of samples between 1.0 and -1.0 if
# samples vary betweenpositive and negative, otherwise
# normalises samples between 1.0 and 0.0
def normalise(data, minV = None, maxV = None):

    if minV is None:
        minV = min(data)

    if maxV is None:
        maxV = max(data)

    if minV >= 0.0:
        return positiveNormalise(data, minV, maxV)


    def normaliseItem(item):
        return ((2.0 * (item - minV)) / (maxV - minV)) - 1.0


    return [normaliseItem(i) for i in data]


# Takes a list of dictionaries containing information on the files
# to plot and information on how they should be plotted, and plots
# them. Returns a list of handles of the plotted graphs
def plotGraphs(plots, minY = None, maxY = None):
    for plot in plots:
        data = np.loadtxt(plot["fileName"], delimiter=",")
        dxs = data[:,0]
        dys = data[:,1]

        ax.plot(dxs, normalise(dys, minY, maxY),
            color = plot["color"], label = plot["label"])



def plotIdealDelay():

    plotsData = [{"fileName": "Laboratory.csv",
                  "color": "red",
                  "label": "Signal"},

                 {"fileName": "Delay5.csv",
                  "color": "green",
                  "label": "5ms Window"},

                 {"fileName": "Delay10.csv",
                  "color": "blue",
                  "label": "10ms Window"},

                 {"fileName": "Delay15.csv",
                  "color": "black",
                  "label": "15ms Window"}
                 ]

    plotGraphs(plotsData)

    pl.title('Ideal Delay')
    pl.ylabel("Normalised S[n]")
    pl.xlabel("Time in Milliseconds")

    pl.grid(True)
    #Place the legend
    box = ax.get_position()
    ax.legend(loc = 'center left', bbox_to_anchor=(1, 0.5))

    pl.savefig('idealDelay.png', bbox_inches='tight')





def plotMovingAverage():

    plotsData = [{"fileName": "Laboratory.csv",
                  "color": "red",
                  "label": "Signal"},

                 {"fileName": "MAverage5.csv",
                  "color": "green",
                  "label": "MA 5ms"},

                 {"fileName": "MAverage10.csv",
                  "color": "blue",
                  "label": "MA 10ms"},

                 {"fileName": "MAverage15.csv",
                  "color": "black",
                  "label": "MA 15ms"}
                ]

    plotGraphs(plotsData)

    pl.title('Moving Average')
    pl.ylabel("Normalised S[n]")
    pl.xlabel("Time in Milliseconds")

    pl.grid(True)

    #Place the legend
    box = ax.get_position()
    ax.legend(loc = 'center left', bbox_to_anchor=(1, 0.5))

    pl.savefig('movingAverage.png', bbox_inches='tight')



def plotConvolution():

    plotData = [{"fileName": "Convolution.csv",
                 "color": "red",
                 "label": "Signal"}]


    plotGraphs(plotData)

    pl.title('Convolution')
    pl.ylabel("Normalised y[n]")
    pl.xlabel("Time in Milliseconds")

    plotGraphs(plotData)
    pl.title('Functions applied to Samples')
    pl.xlabel("Time ms")
    pl.ylabel("Normalised signals S[n]")
    pl.grid(True)
    pl.savefig('convolution.png', bbox_inches='tight')
    pl.legend()




def plotMultiple():
    plotData = [{"fileName": "Laboratory.csv",
           "color":"red",
           "label":"Original Signal"},

           {"fileName": "Energy.csv",
            "color":"green",
            "label":"Energy 30ms"},

           {"fileName": "Magnitude.csv",
            "color":"blue",
            "label":"Magnitude 30ms"},

           {"fileName": "ZeroCrossingRate.csv",
            "color":"cyan",
            "label":"Zero Crossing Rate\n 30ms"}
         ]

    plotGraphs(plotData)
    pl.title('Functions applied to Samples')
    pl.xlabel("Time ms")
    pl.ylabel("Normalised signals S[n]")
    pl.grid(True)

    #Place the legend
    box = ax.get_position()
    #ax.set_position([box.x0, box.y0, box.width * 0.8, box.height])
    ax.legend(loc = 'center left', bbox_to_anchor=(1, 0.5))

    pl.savefig('signals.png', bbox_inches='tight')


def main():
    plotIdealDelay()
    ax.clear()
    plotMovingAverage()
    ax.clear()
    plotConvolution()
    ax.clear()
    plotMultiple()

main()
```

C Lab2 Processing Code (Haskell)

C.1 Lab2.hs

```haskell
module AI.Lab2.Lab2 where

import System.Random
import Data.List
import qualified Data.Vector.Unboxed as VU

-- Sample Data either involves speech
-- or is silent
data SoundType = Speech | Silence
    deriving (Eq, Show)

-- Type alias for
type SampleData = ([Double], SoundType)


loss :: SoundType -> SoundType -> Double
loss expected actual
    | expected == actual = 0
    | otherwise          = 1


-- Calculate mean of a given feature
mean :: [Double] -> Double
mean [] = 0
mean xs = (sum xs) / (fromIntegral $ length xs)


-- Calculate variance of a given feature
variance :: [Double] -> Double
variance [] = 0
variance xs = (1 / sz) * (sum $ map (\x -> (x - mu)^2) xs)
    where
          sz :: Double
          sz = fromIntegral $ length xs

          mu :: Double
          mu = mean xs



-- Computes univariate gaussian from given mean and variance
univariateGaussian :: Double -> Double -> Double -> Double
univariateGaussian mu var x = right / left
    where left = sqrt (2 * pi * var)
          right = exp $ - ((x - mu)^2 / (2 * var))
```

```haskell
-- Calculates gaussian discriminant for univariate gaussian
gaussianDiscriminant :: Double -> Double -> Double -> Double
gaussianDiscriminant mu var x = - log(sqrt(2 * pi * var)) -
                                ((x - mu)^2 / (2 * var)) +
                                log(univariateGaussian mu var x)



-- Calculates the log of the discriminant function
-- for a given SoundType decision.
probability :: SoundType -> [SampleData] -> ([Double] -> Double)
probability st trainData values = sum predictors
    where
         predictors :: [Double]
         predictors = map (\(m, v, vals) ->
                        gaussianDiscriminant m v vals) $ zip3 means vars values

         means :: [Double]
         means = map mean classSamples

         vars :: [Double]
         vars = map variance classSamples

         -- Get the sample data which is of the given SoundType
         classSamples :: [[Double]]
         classSamples = transpose $ map fst $
                            filter (\(values, st') -> st == st') trainData


-- Calculate average loss over the test set after training gaussian
-- discriminant functions with the train set.
averageLoss :: [SampleData] -> [SampleData] -> Double
averageLoss trainData testData =
    (sum $ map (uncurry loss . predict) testData) / (
                fromIntegral $ length testData)

    where
        classPredictors :: [(SoundType, [Double] -> Double)]
        classPredictors = map (\st -> (st, probability st trainData))
                                [Speech, Silence]

        predict :: SampleData -> (SoundType, SoundType)
        predict (features, st) = (predictSt, st)

          -- Select the class with the largest discriminant
          where (predictSt, _) =
                  foldl1 (\(st1, p1) (st2, p2) -> if p2 > p1
                                                  then (st2, p2)
                                                  else (st1, p1)
```

```haskell
                       ) $
                       map (\(st', predictFn) ->
                            (st', predictFn features)) classPredictors


-- Calculate Average loss using cross validation given
-- all test/training sets
cvAverageLoss :: [([SampleData], [SampleData])] -> Double
cvAverageLoss sets =
    (sum $ map (\(trainData, testData) ->
            averageLoss trainData testData) sets) /
                (fromIntegral $ length sets)


-- Given a list of signals, a window length and a function which
-- takes a list of signals and that window lenth and transforms the
-- signal; this function takes the average of the transformation on
-- the signals
averageSig :: [[Int]] -> Int -> (VU.Vector Int -> Int ->
                                   VU.Vector Double) -> [Double]
averageSig signals ms f = avg_f_signals
    where f_signals =
            map (\m -> f (VU.fromList m)
                ms ) signals -- Apply transformation

          sum_f_signals =
            map (VU.foldl' (+) 0) f_signals -- Sum signals

          avg_f_signals =
            map (/ (fromIntegral $ length signals))
                sum_f_signals -- Average signals


-- Given a list of list of signals, a window length and a function which
-- takes a list of signals and that window lenth and transforms the signal;
-- this function takes the log of the average of the transformation on the
-- signals
logAverageSig :: [[Int]] -> Int ->
        (VU.Vector Int -> Int -> VU.Vector Double) -> [Double]

logAverageSig signals ms f = log_avg_f_sig
    where log_avg_f_sig = map log $ averageSig signals ms f


-- Removes kth element from given list, throws an error
-- if the size of the list is >= k
removeK :: [a] -> Int -> (a, [a])
removeK xs = removeK' [] (head xs) (tail xs)
    where removeK' :: [a] -> a -> [a] -> Int -> (a, [a])
          removeK' begin elem end 0 = (elem, begin ++ end)
          removeK’ begin elem end k =
            removeK’ (begin ++ [elem]) (head end) (tail end) (k -1)


-- Attempts to remove k random elements from a given list,
-- returns a tuple containing the list of removed elements and
-- the elements left remaining in the list
-- if total elements <= k then the list itself is returned as
-- the first tuple value
removeRandK :: [a] -> Int -> IO ([a], [a])
removeRandK xs = removeRandK’ ([], xs)
    where
          removeRandK’ :: ([a], [a]) -> Int -> IO ([a], [a])
          removeRandK’ (rnds, []) _ = return (rnds, [])
          removeRandK’ (rnds, xs)  0 = return (rnds, xs)
          removeRandK’ (rnds, xs) k = do
                n <- randomRIO (0, length xs - 1)
                let (elem, xs’) = removeK xs n
                removeRandK’ (elem : rnds, xs’) (k - 1)


-- Splits data into k uniformly sized random subsets
-- requires that k divides the length of the
-- given data, and that the length of data is > 0
-- otherwise an error is thrown
splitK :: [a] -> Int -> IO [[a]]
splitK [] _ = error "splitK: given list is empty"
splitK xs k =
    if length xs ‘mod‘ k /= 0
        then error $ "splitK: " ++ show k ++ " does not divide " ++
                        show (length xs)

        else splitK’ ([], xs) k

    where
          splitK’ :: ([[a]], [a]) -> Int -> IO [[a]]
          splitK’ (splits, []) 0 = return splits
          splitK’ (splits, xs) k = do
            (set, leftOver) <- removeRandK xs (length xs ‘div‘ k)
            splitK’ (splits ++ [set], leftOver) (k - 1)


-- From partitioned data obtain all training and test sets
getTrainTest :: [[a]] -> [([a], [a])]
getTrainTest xs =
    map (\k ->
        let (test, train) = removeK xs k
            in (concat train, test)) [0 .. length xs - 1]
```

C.2 Main.hs

```haskell
import System.IO
import System.Directory
import Data.List
import System.FilePath
import qualified Data.Vector.Unboxed as VU

import AI.Lab2.Lab2
import AI.Lab1.Lab1


samplesDir :: String
samplesDir = "samples"


-- Obtain samples from file
getSamples :: String -> IO [Int]
getSamples fPath = do
    handle <- openFile fPath ReadMode
    contents <- hGetContents handle

    -- Obtain a list of samples (S[n]) as strings from file
    let strSamples = lines contents

    -- Convert samples to integers
    let samples = map (\s -> (read s :: Int)) strSamples

    return samples


-- Given a filename and a list of values, writes
-- the given values to the file. One per each line.
writeDat :: String -> [Double] -> IO ()
writeDat fName signals = writeFile (fName ++ ".dat") $
                            unlines $ map show signals



main :: IO ()
main = do
    -- Obtain absolute paths to all files in the given folder
    folderFiles <- getDirectoryContents samplesDir >>=
                    mapM (canonicalizePath . (samplesDir </>))

    -- Filter to obtain paths for only the sample files
    let sampleFiles = filter (isSuffixOf ".dat") folderFiles
    let speechSampleFiles = filter (isInfixOf "speech") sampleFiles
    let silenceSampleFiles = filter (isInfixOf "silence") sampleFiles
```

```
-- Extract samples from all the files
speechSamples <- mapM getSamples speechSampleFiles
silenceSamples <- mapM getSamples silenceSampleFiles

let samples = speechSamples ++ silenceSamples

-- Log of Average Short Term Energy
let logAvgEnergy = logAverageSig samples win_sz_ms energy
writeDat "log_avg_ste" logAvgEnergy

-- Log of Average Short Term Magnitude Signal
let logAvgMtude = logAverageSig samples win_sz_ms magnitude
writeDat "log_avg_stm" logAvgMtude

-- Average Zero Crossing Rate Signal
let avgZCR = averageSig samples win_sz_ms zeroCrossingRate
writeDat "avg_zcr" avgZCR

let speechVars = [logAverageSig speechSamples win_sz_ms energy,
                  logAverageSig speechSamples win_sz_ms magnitude,
                  averageSig speechSamples win_sz_ms zeroCrossingRate]

let speechData = map (\d -> (d, Speech)) $
        transpose speechVars

let silenceVars = [logAverageSig silenceSamples win_sz_ms energy,
                   logAverageSig silenceSamples win_sz_ms magnitude,
                   averageSig silenceSamples win_sz_ms zeroCrossingRate]

let silenceData = map (\d -> (d, Silence)) $
        transpose silenceVars

-- Randomly split speech and silence data into
-- 10 equally sized disjoint subsets
speechSplits <- splitK speechData 10
silenceSplits <- splitK silenceData 10

-- Concatonate each silence and speech splits, obtain the training/test
-- sets and calculate the average loss using a 10-fold cross
-- validation.
let splits = zipWith (++) speechSplits silenceSplits

let trainTest = getTrainTest splits

let avgLoss = cvAverageLoss trainTest

print avgLoss
where
    win_sz_ms = 15
```

D Lab2 Plotting Code (Python)

```python
#Plots the same data generated by the Haskell program
import matplotlib.pyplot as pl
import numpy as np


def main():
    energies = np.loadtxt("log_avg_ste.dat")
    magnitudes = np.loadtxt("log_avg_stm.dat")
    zcrs = np.loadtxt("avg_zcr.dat")

    pl.scatter(energies, magnitudes, color = "red")
    pl.title("Energy against Magnitude")
    pl.xlabel("Log average of energy")
    pl.ylabel("Log average of magnitudes")
    pl.savefig("eam.png", bbox_inches='tight')

    pl.scatter(energies, zcrs, color = "red")
    pl.title("Energy against Zero Crossing Rate")
    pl.xlabel("Log of Average of Energy")
    pl.ylabel("Average Zero Crossing Rate")
    pl.savefig("eac.png", bbox_inches='tight')


    pl.scatter(magnitudes, zcrs, color = "red")
    pl.title("Magnitude against Zero Crossing Rate")
    pl.xlabel("Log of average of Magnitudes")
    pl.ylabel("Average of Zero Crossing Rates")
    pl.savefig("maz.png", bbox_inches='tight')

main()
```