Narges Sadat Seyed Haeri- 810100165

CA4 Report

- ## *Part 1:*

1.1: we did this part in CA3 too without further explanation we create the map set with the code below:

```
%% 1.1 loading mapset

Alphabet='abcdefghijklmnopqrstuvwxyz .,!;"';

num_alphabet=length(Alphabet);
mapset=cell(2,num_alphabet);
for i=1:num_alphabet
    mapset{1,i}=Alphabet(i);
    mapset{2,i}=dec2bin(i-1,5);
end
char_bin_len = length(mapset{2, 1});


fs = 100;
```

1.2:

This MATLAB function, coding_amp, performs amplitude modulation to encode a binary message at a specified bit rate into an analog signal. The function takes two parameters: binary_msg, the binary message to be encoded, and bit_rate, the rate at which bits are encoded per second.

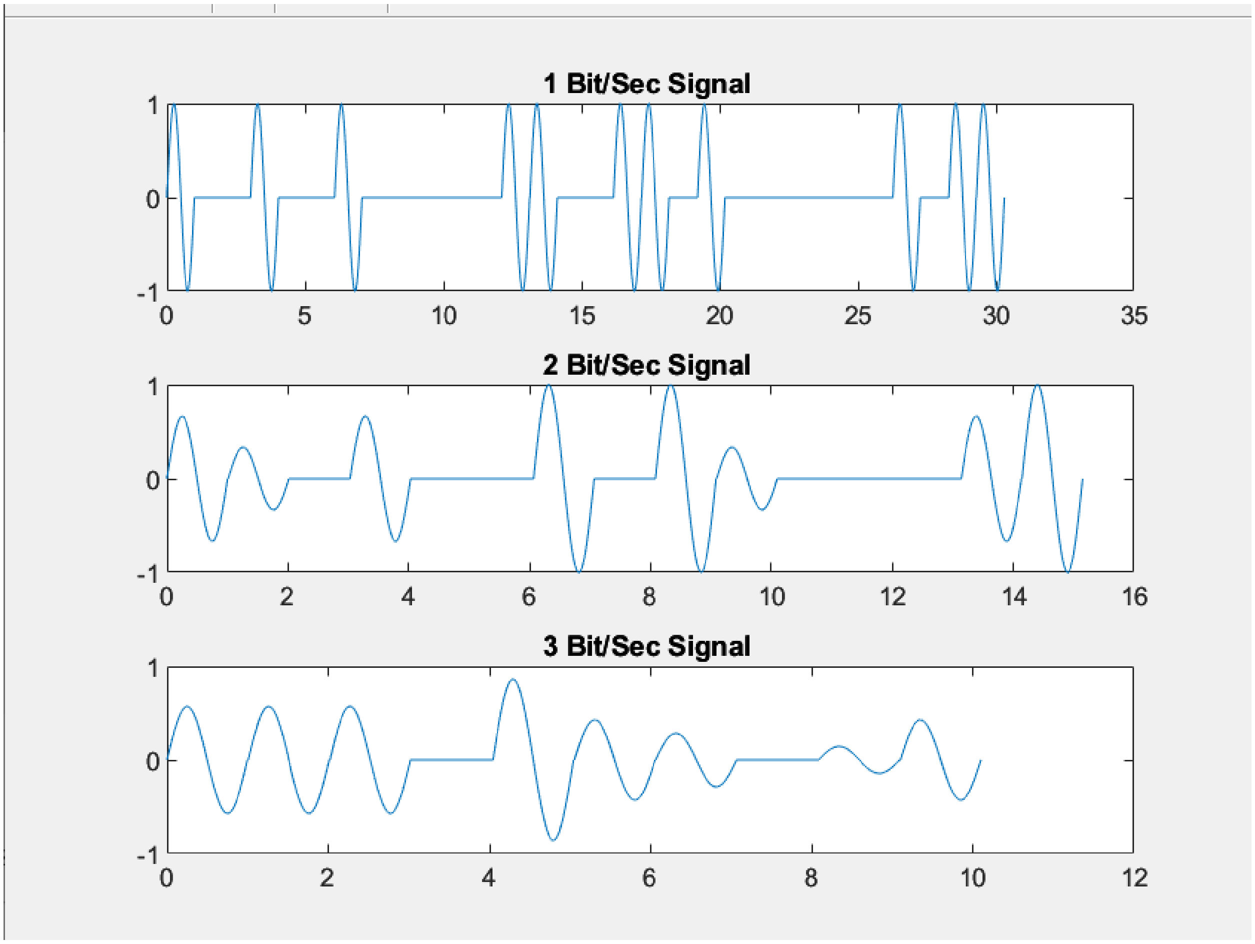
The function initializes parameters such as the sampling frequency (fs), time step (step), and signal duration (end_time). It then iterates through the binary message, converting each group of bits (according to the specified bit rate) to a decimal coefficient. For each coefficient, it generates a sine wave modulated by that coefficient and appends it to the coded_signal array.

The resulting coded_signal is a sequence of amplitude-modulated sine waves representing the binary message. The function returns this coded signal for further use or analysis. Note that the signal duration is set to 1 second (end_time=1), and the function assumes a sampling frequency of 100 Hz (fs=100), which can be adjusted based on your requirements.

```
function signal=coding_amp(binary_msg,bit_rate)

fs=100;
step=1/fs;
end_time=1;

coded_signal=[];
for i=1:bit_rate:length(binary_msg)
    Coefficient=bin2dec(binary_msg(i:i+bit_rate-1))/(2^bit_rate-1);
    t=0:step:end_time;
    y=Coefficient.*sin(2*pi*t);
    coded_signal=[coded_signal y];
end

signal=coded_signal;
end
```

This code converts the message 'signal' into a binary format using a predefined character-to-binary map mapset. It then visualizes the coded signals at different bit rates (1, 2, and 3 bits per second) using amplitude modulation. The code iterates through each character in the message, finds its corresponding binary representation in the mapset, and concatenates the binary values. The resulting binary message is encoded into a signal at different bit rates using the coding_amp function. Subsequently, the coded signals are plotted in separate subplots, with each subplot representing a different bit rate. The time vector t is generated for plotting the signals over time.

```
msg = 'signal';
index=[];
for i=1:length(msg)
    ch=msg(i);
    index=[index, find(strcmp(ch,mapset(1,:))==1)];
end
bin_msg=cell2mat(mapset(2,index));

figure

bit_rates = [1, 2, 3];
for j = 1:length(bit_rates)
    bit_rate = bit_rates(j);
    subplot(length(bit_rates), 1, j);
    coded_signal = coding_amp(bin_msg, bit_rate);
    t = linspace(0, length(coded_signal) / fs, length(coded_signal));
    plot(t, coded_signal);
    title([num2str(bit_rate),' Bit/Sec Signal ']);
end
```



1.3:

This function, decoding_amp, performs amplitude demodulation to decode a modulated signal into a binary message at a specified bit rate. The function takes two parameters: signal, the amplitude-modulated signal to be decoded, and bit_rate, the rate at which bits were originally encoded per second.

The function initializes parameters such as the sampling frequency (Fs), time step (step), signal duration (end_time), and chunk size (chunk). It also sets a threshold based on the maximum possible value for the given bit rate.

A loop iterates through the signal in chunks, and for each chunk, it calculates the cross-correlation with a reference sine wave. The correlation value is compared with the threshold, and if it exceeds the threshold, the corresponding binary representation of the coefficient is appended to the result.

The function returns the decoded binary message (binary_msg). the function uses the xcorr function for cross-correlation and assumes a threshold value of 2*(2^bit_rate-1). The sampling frequency is set to 100 Hz (Fs=100), and the signal duration is 1 second (end_time=1). Adjust these parameters based on your specific requirements.

```matlab
function binary_msg=decoding_amp(signal,bit_rate)
Fs=100;
step=1/Fs;
end_time=1;
bin='';
chunk = round(end_time * Fs);
threshold = 2*(2^bit_rate-1);

Coefficient=[];
for i=0:1:(2^bit_rate-1)
    curr_coff=i/(2^bit_rate-1);
    Coefficient = [Coefficient curr_coff];
end

for i = 1:chunk:(length(signal) - chunk)
    sig_part = signal(i:i+chunk-1);
    t=0:step:end_time;
    Correlation = max(0.01*xcorr(sig_part, 2*sin(2*pi*t)));
    [th, index] = min(abs(Coefficient - Correlation));

     if threshold > th
        bin=strcat(bin,dec2bin(index-1,bit_rate));
     end
end
```

we systematically explored the encoding and decoding of a binary message at different bit rates. We employed a loop to iterate through a set of specified bit rates (bit_rates). For each bit rate, we encoded the original binary message (bin_msg) into an amplitude-modulated signal using the coding_amp function. Subsequently, we decoded this signal back into a binary message at the same bit rate using the decoding_amp function. To bring it full circle, we converted the decoded binary message into a string representation using a predefined character-to-binary map (mapset) and displayed the result. This iterative process allowed us to observe the impact of varying bit rates on the accuracy of information recovery during encoding and decoding.

```matlab
%% 1.3 decoding a msg

for i = 1:length(bit_rates)
    bit_rate = bit_rates(i);
    coded_signal = coding_amp(bin_msg, bit_rate);
    binary_decoded_signal = decoding_amp(coded_signal,bit_rate);
    decoded_signal = binary_to_string(binary_decoded_signal, mapset);
    disp(['the encoded massage (with the bitrate =',num2str(bit_rate),')is :', decoded_signal])
end
```

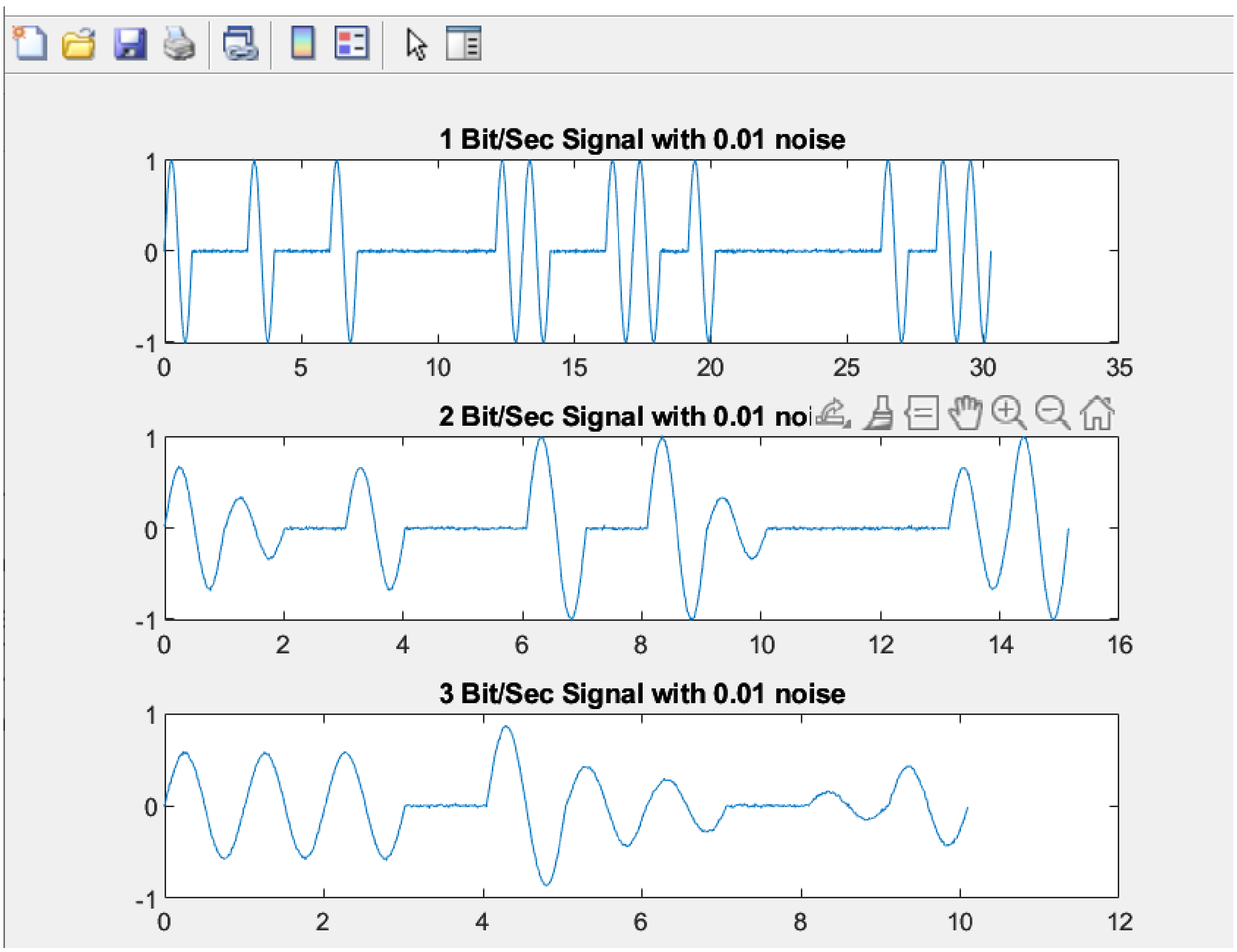
```
>> p1
the encoded massage (with the bitrate =1)is :signal
the encoded massage (with the bitrate =2)is :signal
the encoded massage (with the bitrate =3)is :signal
```

1.6:

Adding noise to the decoding msg code with randn func:

```
the encoded massage (with the bitrate =3)is :signal
the encoded massage (with the bitrate =1 and noise =0.01)is :signal
the encoded massage (with the bitrate =2 and noise =0.01)is :signal
the encoded massage (with the bitrate =3 and noise =0.01)is :signal
mean: 0.018121
```

```matlab
%% 1.4 Adding noise

figure
noise = 0.6;

for i = 1:length(bit_rates)
    bit_rate = bit_rates(i);
    coded_signal = coding_amp(bin_msg, bit_rate);
    noisy_signal = coded_signal + noise * randn(size(coded_signal));
    binary_decoded_signal = decoding_amp(noisy_signal, bit_rate);
    decoded_signal = binary_to_string(binary_decoded_signal, mapset);
    disp(['the encoded massage (with the bitrate =',num2str(bit_rate),' and noise =',num2str(noise),')is :', decoded_signal])
    subplot(length(bit_rates), 1, i);
    t = linspace(0, length(coded_signal) / fs, length(coded_signal));
    plot(t, noisy_signal);
    title([num2str(bit_rate),' Bit/Sec Signal with ',num2str(noise),' noise']);

end
```

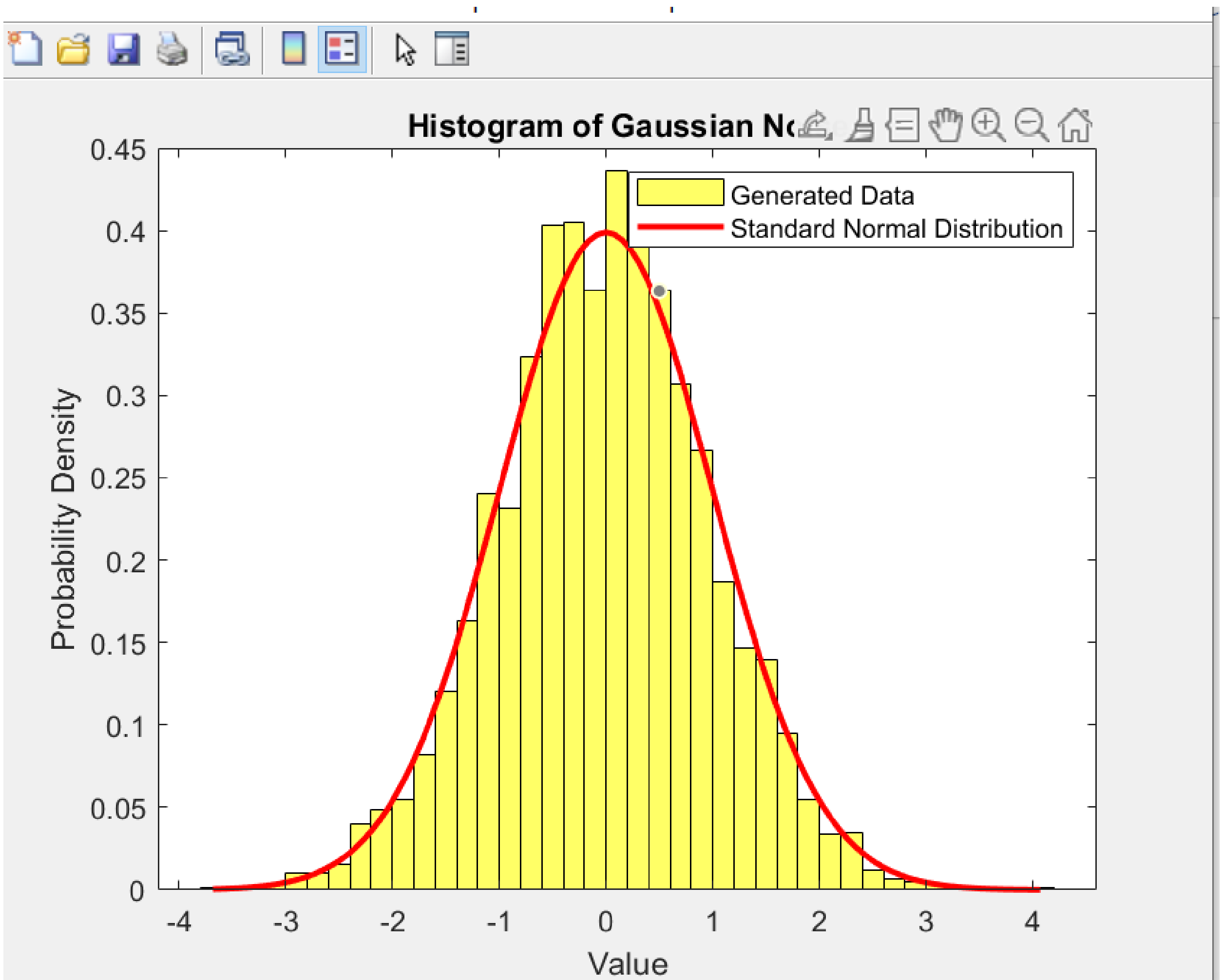


1.5:

Here is the code to plot the signal :

```
%% 1.5
noisySignal = randn(1,3000);
mu=abs(mean(noisySignal));
disp(['mean: ', num2str(mu)]);
disp(['variance: ', num2str((std(noisySignal))^2)]);

figure;
histogram(noisySignal, 'Normalization', 'pdf', 'EdgeColor', 'black', 'FaceColor', 'yellow');
hold on;

x = linspace(min(noisySignal), max(noisySignal), 100);
y = normpdf(x, 0, 1);
plot(x, y, 'LineWidth', 2, 'Color', 'red');

title('Histogram of Gaussian Noise');
xlabel('Value');
ylabel('Probability Density');
legend('Generated Data', 'Standard Normal Distribution');
```

The histogram is compatible with SND:

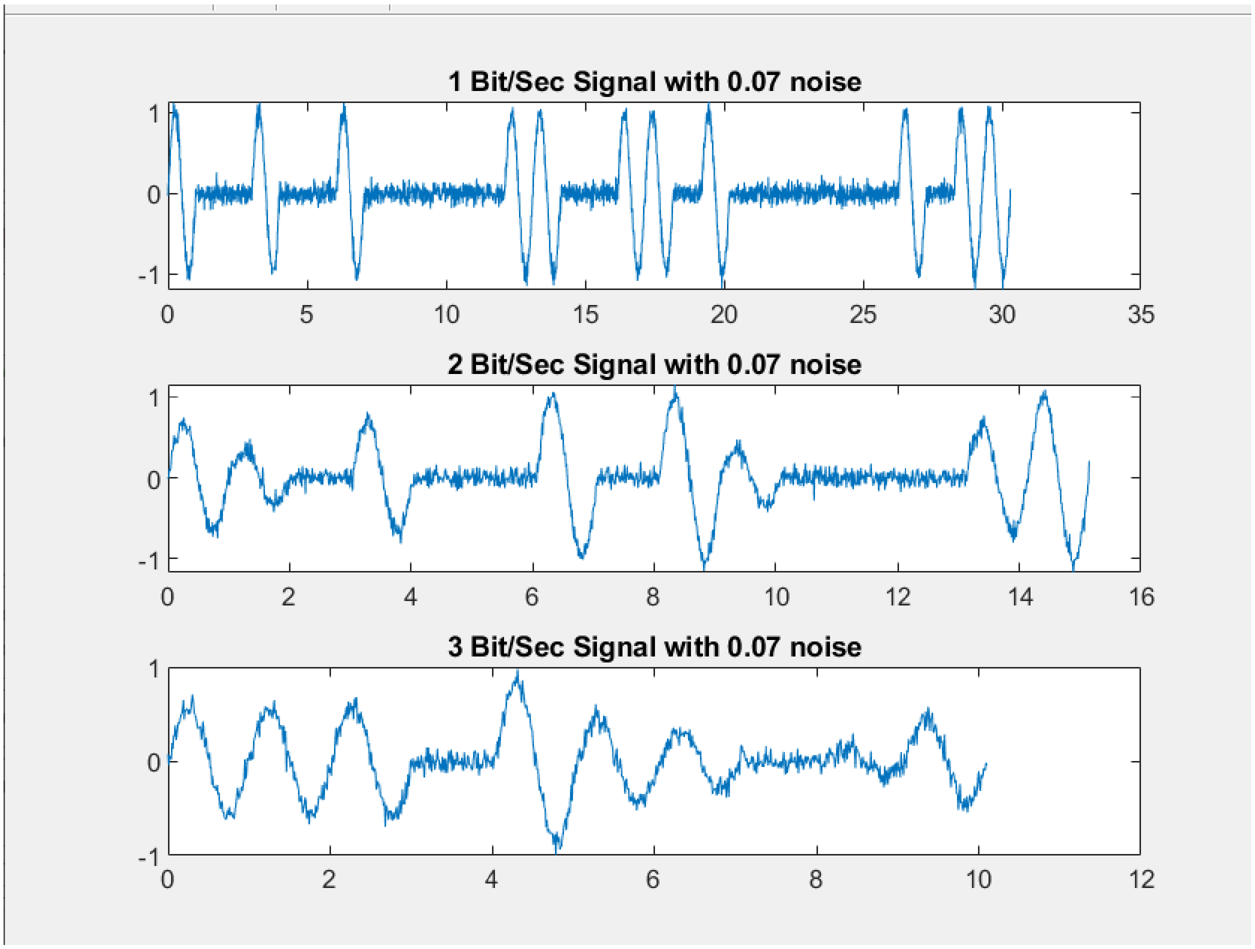


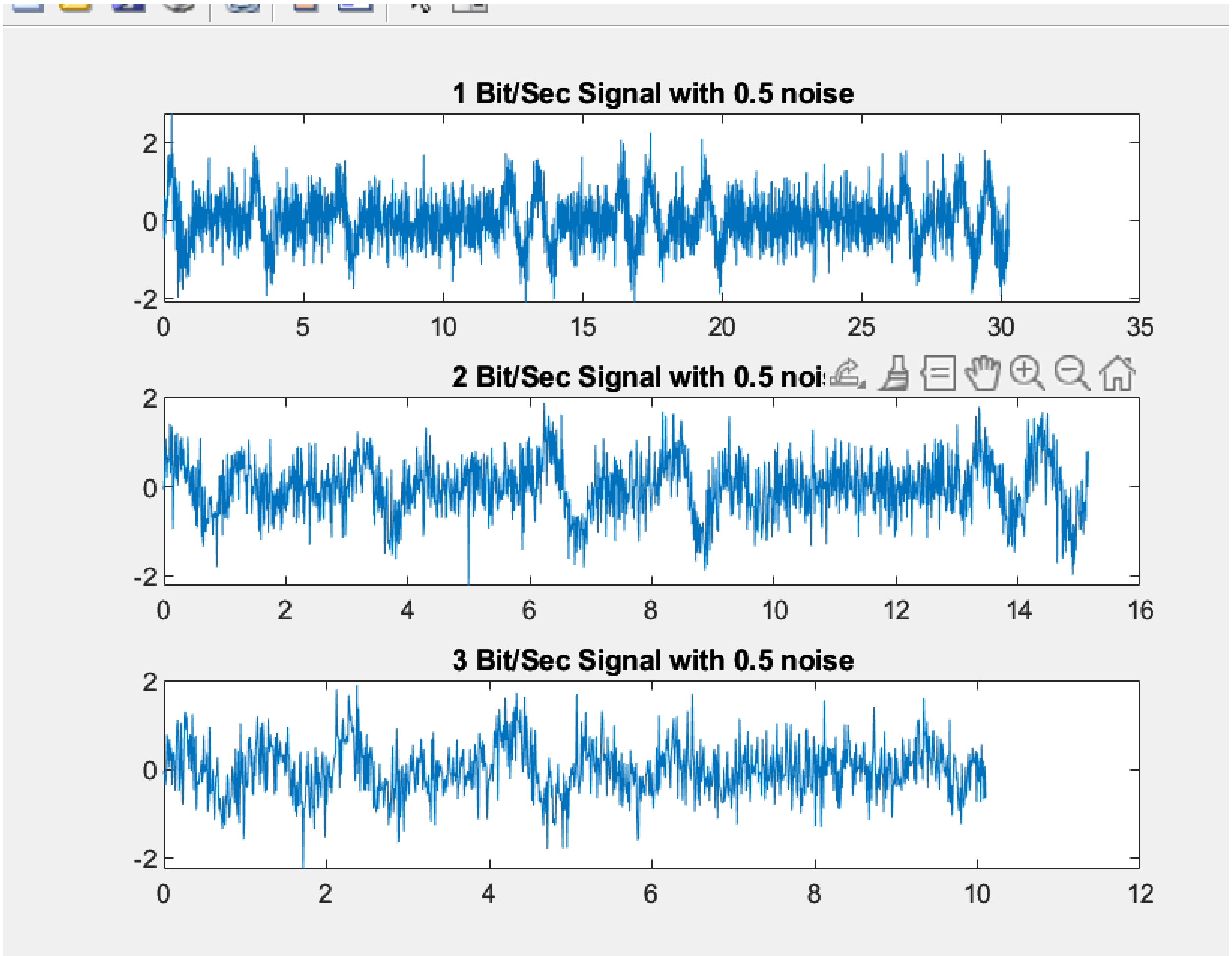
we can see that the mean is near to zero and the variance is near 1:

```
mean: 0.018121
variance: 0.9603
```

1.7:

The signal with bitrate of 1 is the strongest.





1.8:

Until noise = 0.8 the signal with bitrate 1 was giving the correct answer for signal with bit rate 2 it takes until noise =0.6 and for bitrate 3 it was until noise =0.4

1.10:

the text content is kept short, they can provide the entire thing within one second. However, if not, due to computational constraints, the processing speed may decrease.

1.11:

We can’t do it because the noise gets stronger too.

1.12:

Between 2 to 24 MB per second.

- ***Part 2:***

  This code simulates a basic radar system for distance estimation. It begins by defining parameters such as time step (ts), total simulation time (T), pulse duration (tau), speed of light (C), target distance (R), and reflection coefficient (alpha). It generates a transmitted signal, simulates signal reflection from a target at distance R, and computes the convolution of the transmitted and received signals to estimate the target distance. The code then introduces noise to the received signal, repeats the distance estimation process, and checks the accuracy of the estimation. The accuracy is assessed by comparing the calculated distance with the known target distance (R), and a message is displayed indicating whether the estimation is accurate within a 10-meter margin. The code provides insight into how noise affects distance estimation in radar systems.

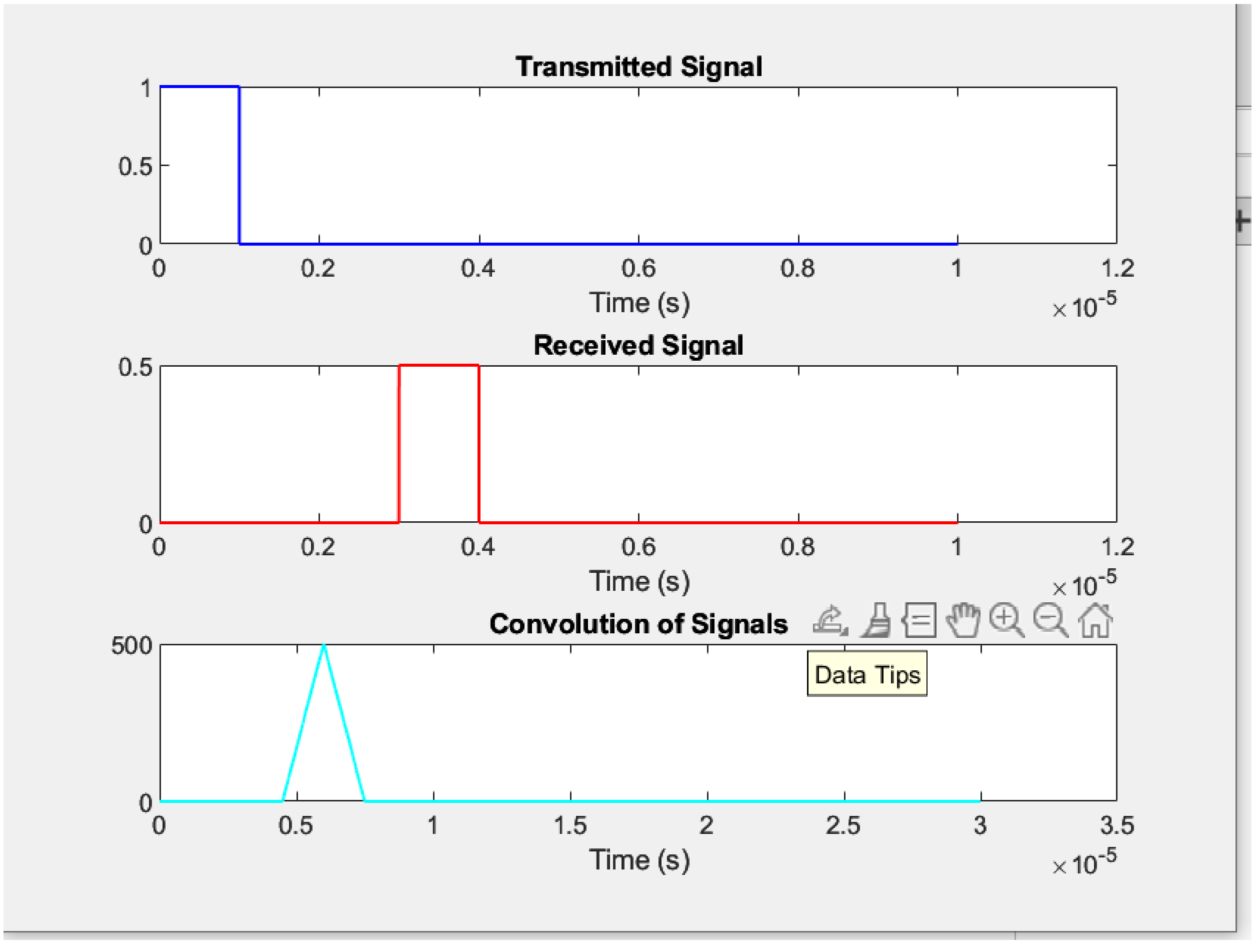
```
N = round(tau / ts);
sentSignal = zeros(size(t));
sentSignal(1 : N) = 1;
td_index = round(td / ts);
receivedSignal = zeros(size(t));
receivedSignal(td_index : td_index + N) = alpha;


figure;

subplot(3, 1, 1);
plot(t, sentSignal, 'b', 'LineWidth', 1);
title('Transmitted Signal');
xlabel('Time (s)');

subplot(3, 1, 2);
plot(t, receivedSignal, 'r', 'LineWidth', 1);
title('Received Signal');
```

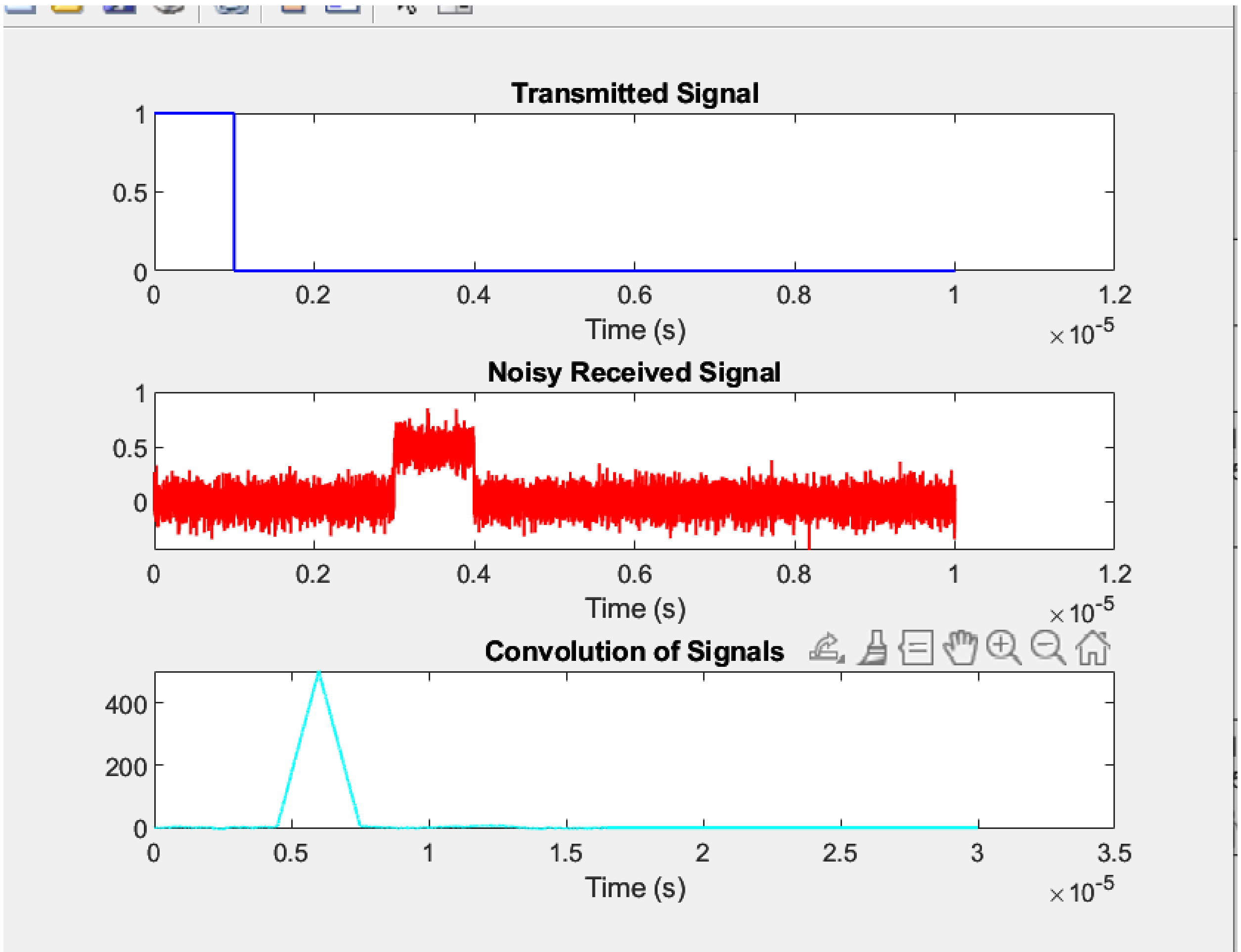
```
%% 2.2
co = conv(sentSignal, receivedSignal);
t_conv = linspace(0, T + (length(co) - 1) * ts, length(co));
subplot(3, 1, 3);
plot(t_conv, co, 'c', 'LineWidth', 1);
title('Convolution of Signals');
xlabel('Time (s)');
[MAXCO, td1] = max(co);
td1 = td1*ts - tau;
R1 = td1 * C / 2;
fprintf('Calculated Distance: %.2f meters\n', R1);

```

```
%% 2.3
noisePower = 0.1;
noisyReceivedSignal = receivedSignal + noisePower * randn(size(receivedSignal));
figure;

subplot(3, 1, 1);
plot(t, sentSignal, 'b', 'LineWidth', 1);
title('Transmitted Signal');
xlabel('Time (s)');

subplot(3, 1, 2);
plot(t, noisyReceivedSignal, 'r', 'LineWidth', 1);
title('Noisy Received Signal');
xlabel('Time (s)');

co = conv(sentSignal, noisyReceivedSignal);
t_conv = linspace(0, T + (length(co) - 1) * ts, length(co));
subplot(3, 1, 3);
```


```
co = conv(sentSignal, noisyReceivedSignal);
t_conv = linspace(0, T + (length(co) - 1) * ts, length(co));
subplot(3, 1, 3);
```


```
plot(t_conv, co, 'c', 'LineWidth', 1);
title('Convolution of Signals');
xlabel('Time (s)');

[MAXCO, td1] = max(co);
td1 = td1*ts - tau;
R1 = td1 * C / 2;
fprintf('Calculated Distance for noisy signal: %.2f meters\n', R1);

if abs(R1 - R) < 10
    disp('Distance estimation is accurate within 10 meters.');
else
    disp('Distance estimation error is greater than 10 meters.');
end
```


Transmitted Signal
Time (s)
Received Signal
Time (s)
Convolution of Signals
Data Tips
Time (s)




```
Calculated Distance: 449.85 meters
Calculated Distance for noisy signal: 450.00 meters
Distance estimation is accurate within 10 meters.
fx >> 
```

- ***Part 3:***

We first create the mapset:

```
numberVoices = containers.Map('KeyType', 'double', 'ValueType', 'any');

numbers = [1:19, 20, 30, 40, 50, 60, 70, 80, 90];

for number = numbers

    filename =[num2str(number) '.m4a'];

    [voice, sampleRate] = audioread(filename);

    numberVoices(number) = struct('voice', voice, 'sampleRate', sampleRate);
end


[voice, sampleRate] = audioread('the_number.m4a');
numberVoices(28) = struct('voice', voice, 'sampleRate', sampleRate);

[voice, sampleRate] = audioread('to_counter.m4a');
numberVoices(29) = struct('voice', voice, 'sampleRate', sampleRate);

[voice, sampleRate] = audioread('o.m4a');
numberVoices(30) = struct('voice', voice, 'sampleRate', sampleRate);


save('numberVoicesMapset.mat', 'numberVoices');
```

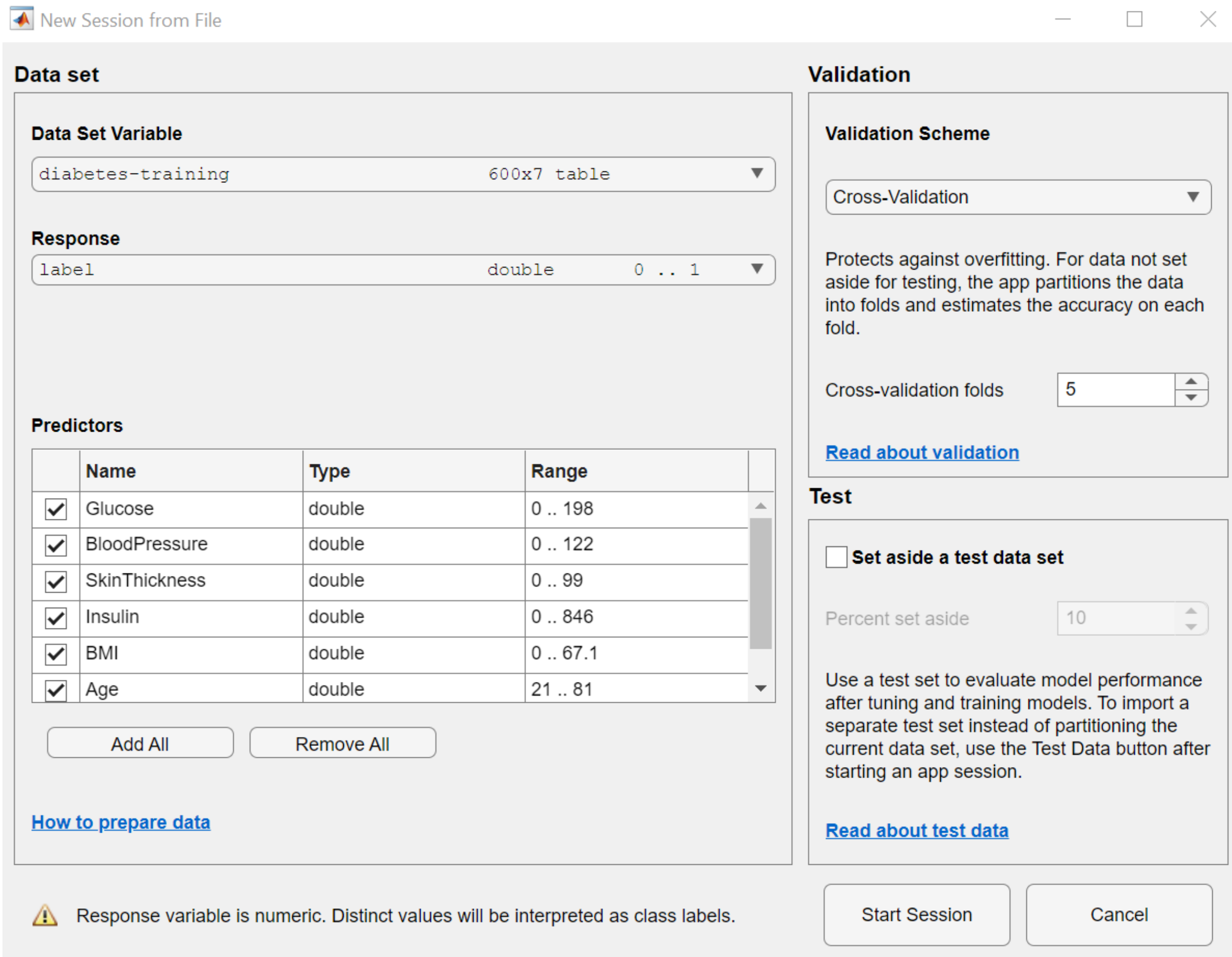
Then we use it in p3:

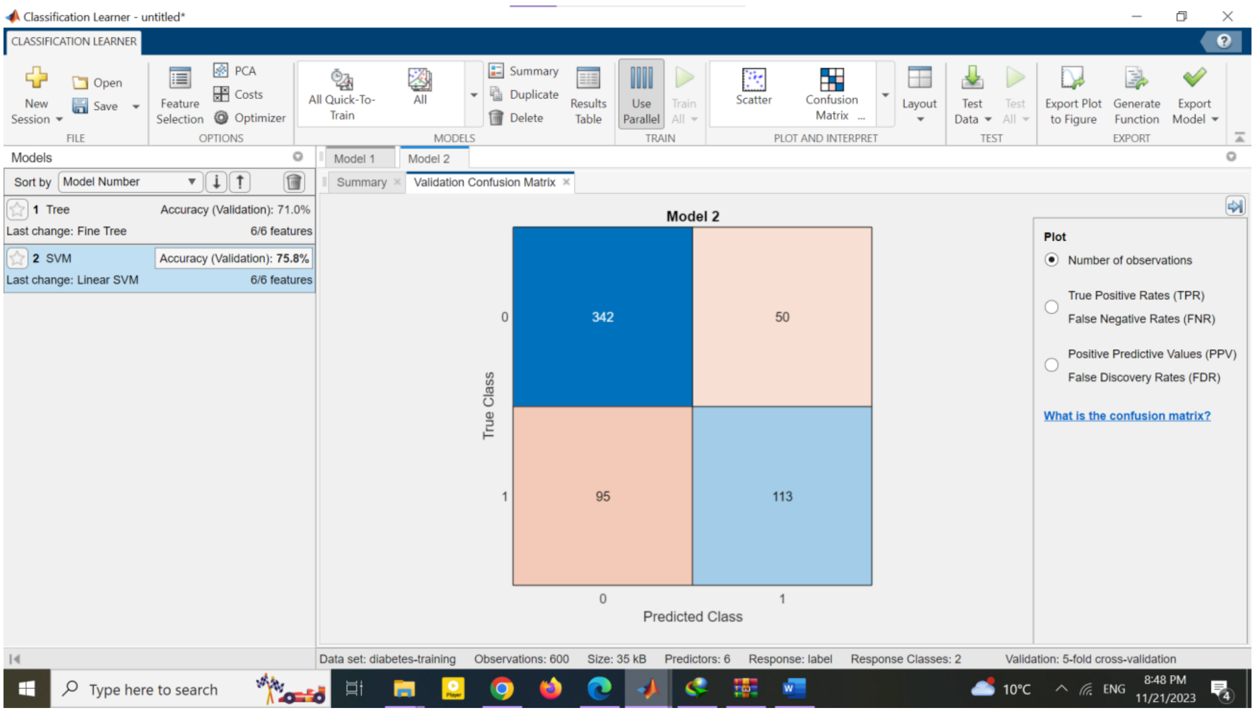
```
function calling_customer(number1, number2)

    load('numberVoicesMapset.mat', 'numberVoices');

    the_number = numberVoices(28).voice;
    to_counter = numberVoices(29).voice;


    if (number1 < 20) || (mod(number1, 10) == 0)
        voice1 = numberVoices(number1).voice;
        voice2 = numberVoices(number2).voice;
        combinedVoice = [the_number;voice1;to_counter; voice2];
    else
        num1=number1 - mod(number1, 10);
        voice1 = numberVoices(num1).voice;
        voice2 = numberVoices(number2).voice;
        o_sound = numberVoices(30).voice;
        number3 = mod(number1, 10);
        voice3 = numberVoices(number3).voice;
        combinedVoice = [the_number;voice1;o_sound;voice3;to_counter; voice2];
    end


    sound(combinedVoice, numberVoices(number2).sampleRate);
end
```

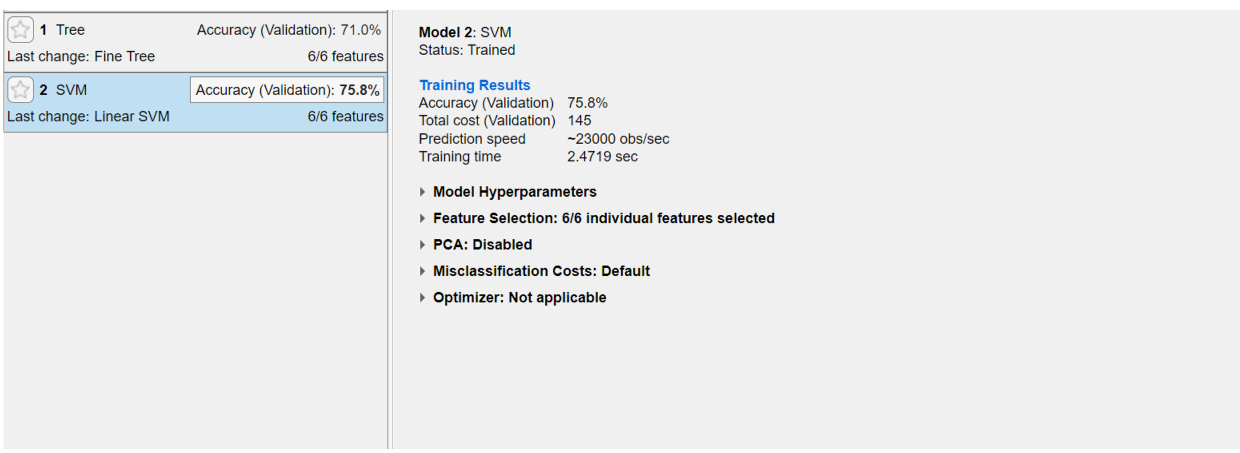
- ***Part 4:***

4.1:

The accuracy validation is %75.

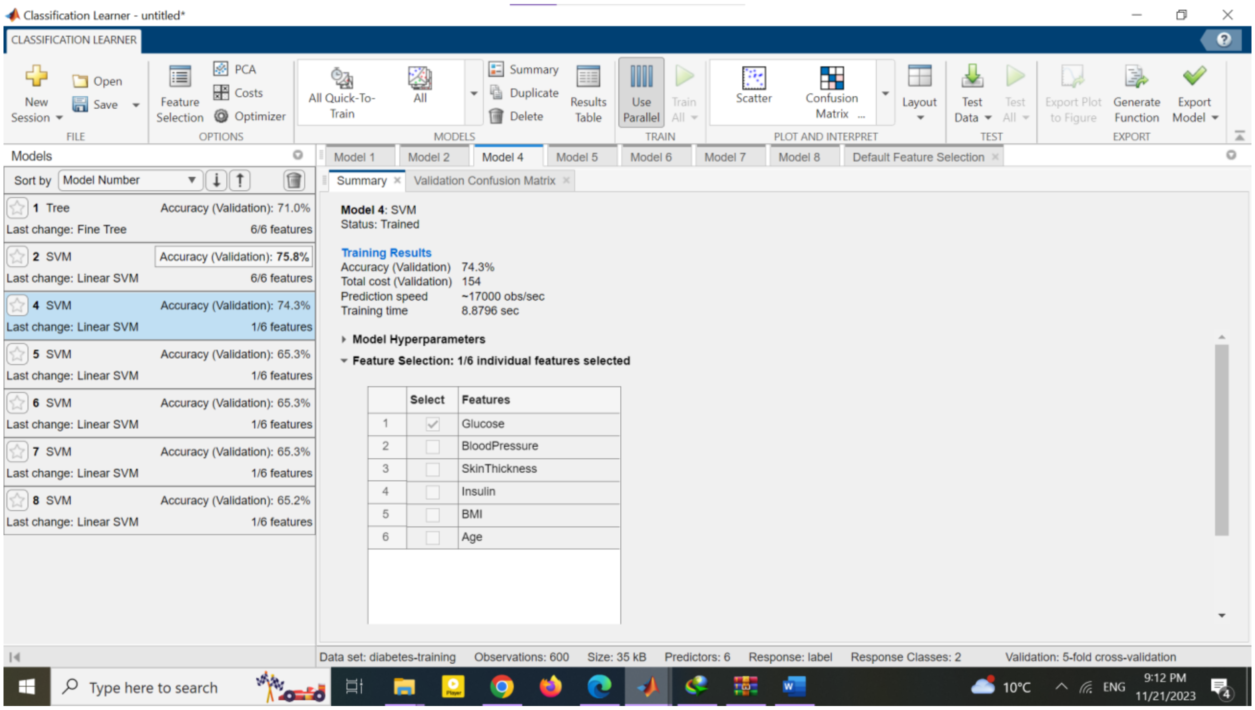




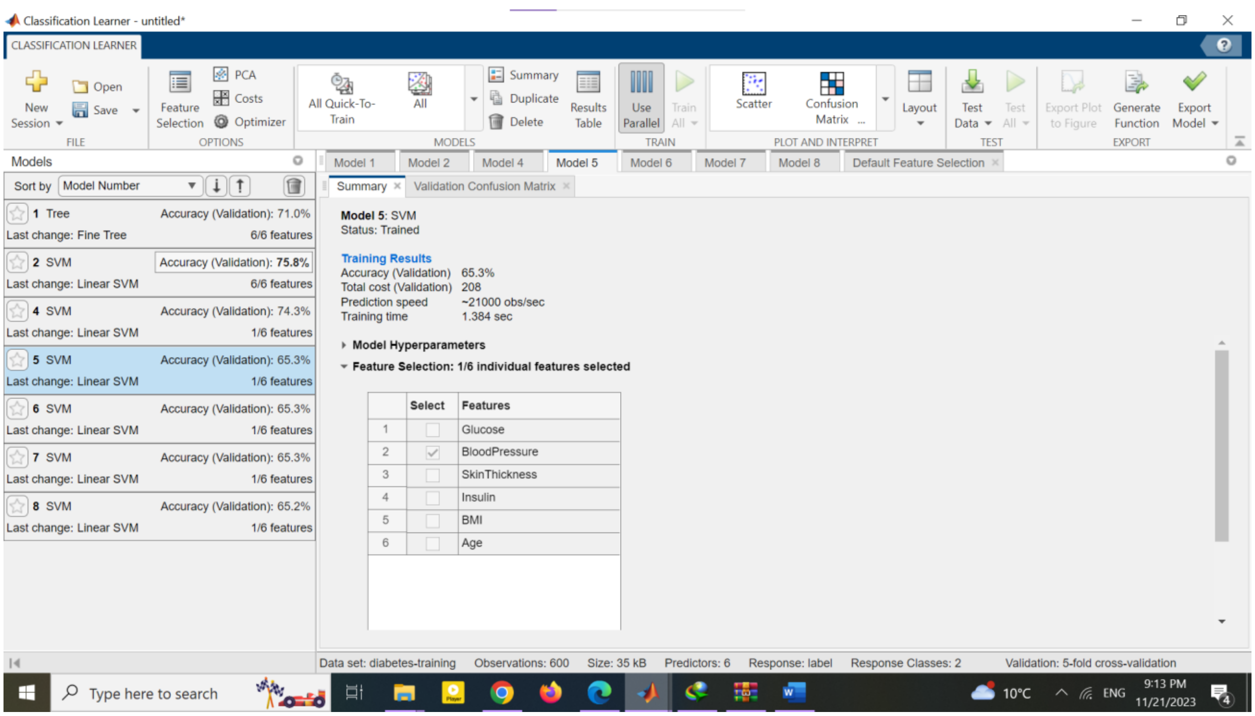


4.2:

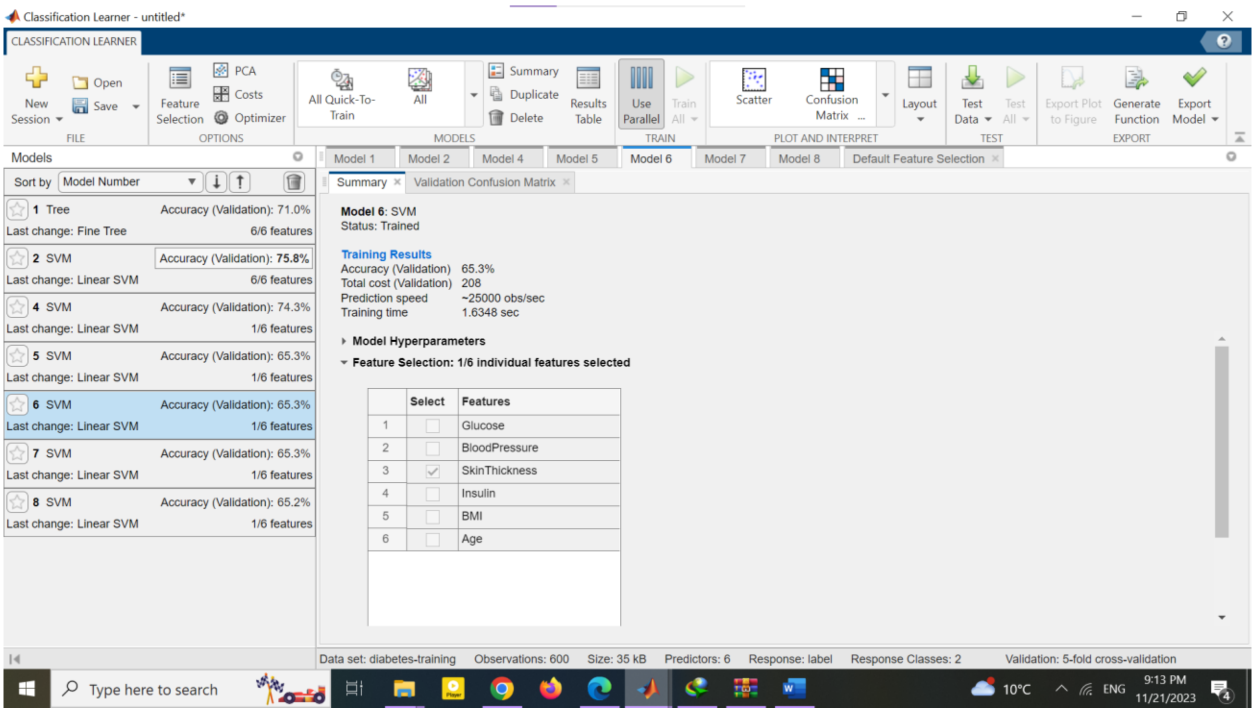
For glucose:



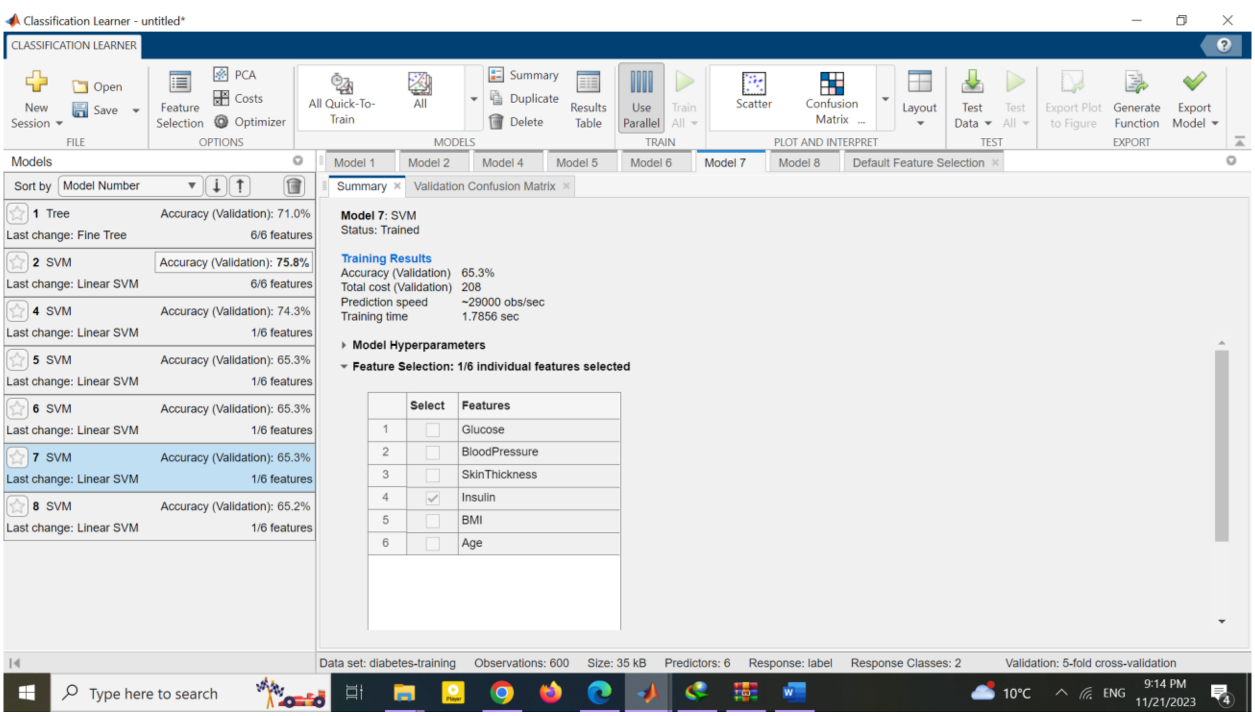
For blood pressure:



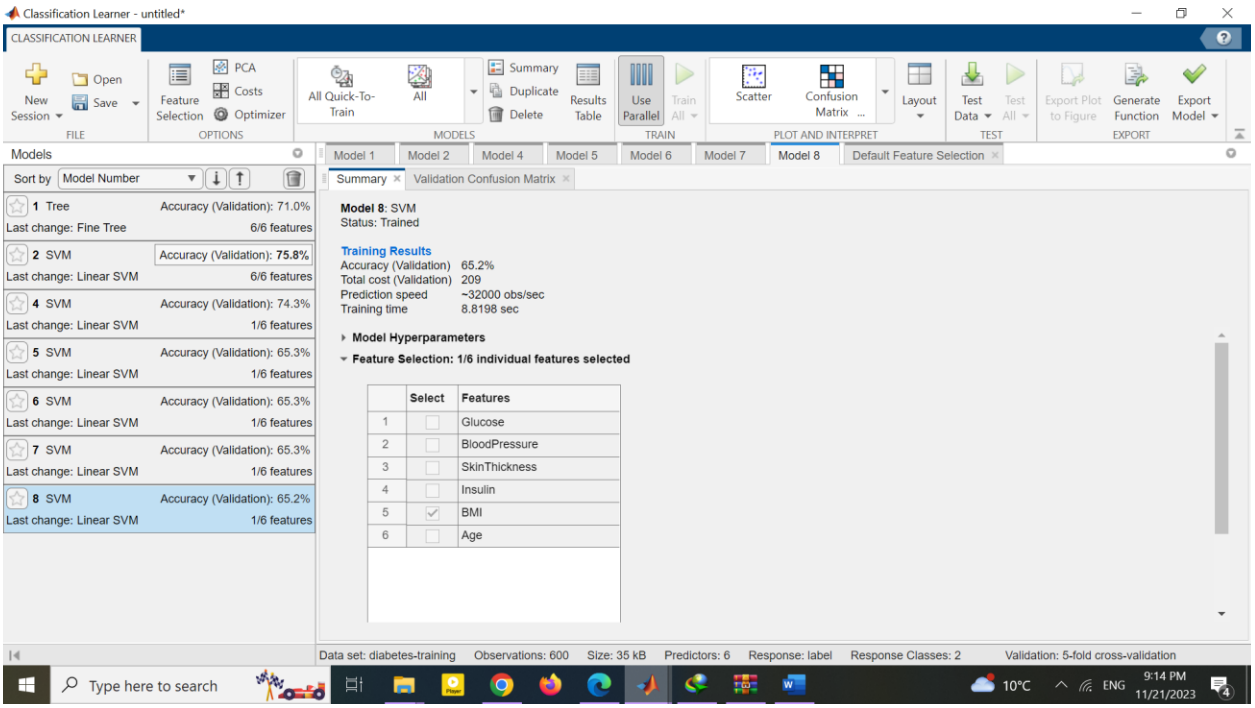
For skin thickness:



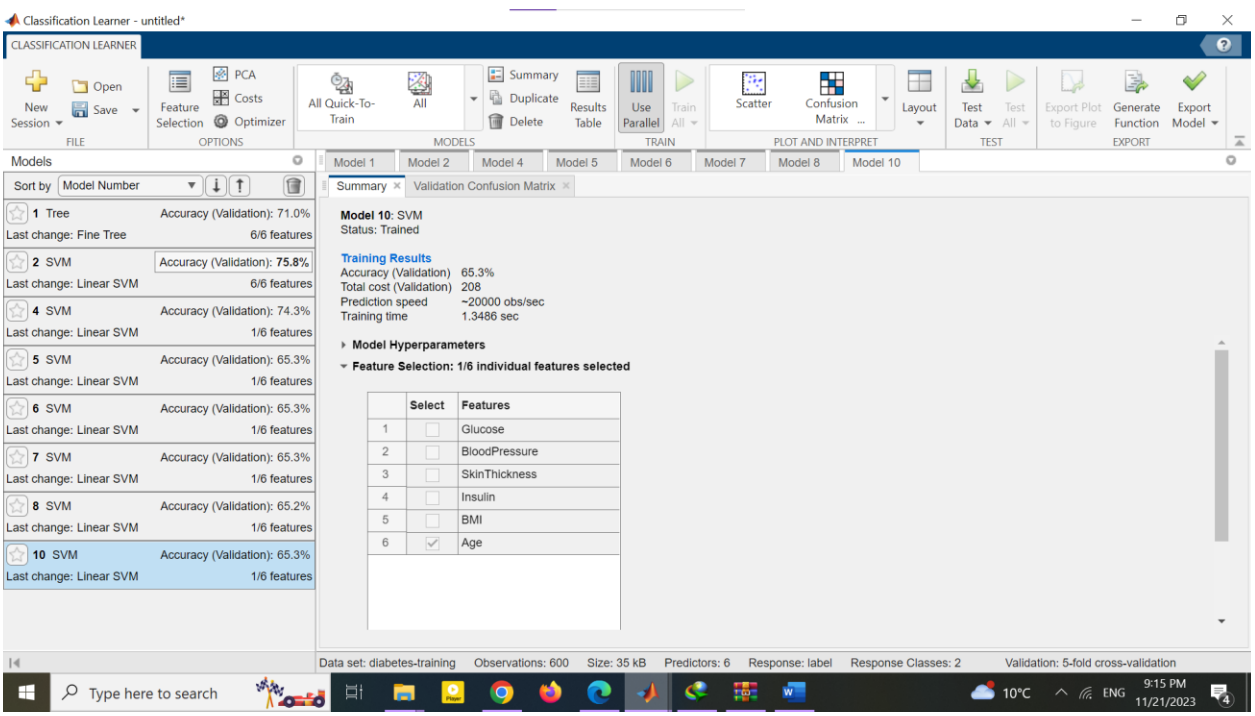
For insulin:



For BMI:



For age:



Therefore, glucose is the most accurate one.

4.3 and 4.4 :

We estimate accuracy of the data in the table with the code below:

```
%% 4.3
table = readtable('diabetes-training.csv');
predictedLabel=trainedModel.predictFcn(table);

actualLabels = table2array(table(:, end));

comparison = predictedLabel == actualLabels;

similarityPercentage = sum(comparison) / length(comparison) * 100;

disp(['Percentage of Similarity: ', num2str(similarityPercentage), '%']);

%% 4.4
table = readtable('diabetes-validation.csv');
predictedLabel2=trainedModel.predictFcn(table);

actualLabels1 = table2array(table(:, end));

comparison = predictedLabel2 == actualLabels1;

similarityPercentage = sum(comparison) / length(comparison) * 100;

disp(['Percentage of Similarity: ', num2str(similarityPercentage), '%']);
```

Here are the results:

```
>> p4
Percentage of Similarity: 77.5%
Percentage of Similarity: 78%
```