MLPR Assignment 1 2018

s1879797, s1873947

November 9, 2018

1 Question 1

1.1 Q1a

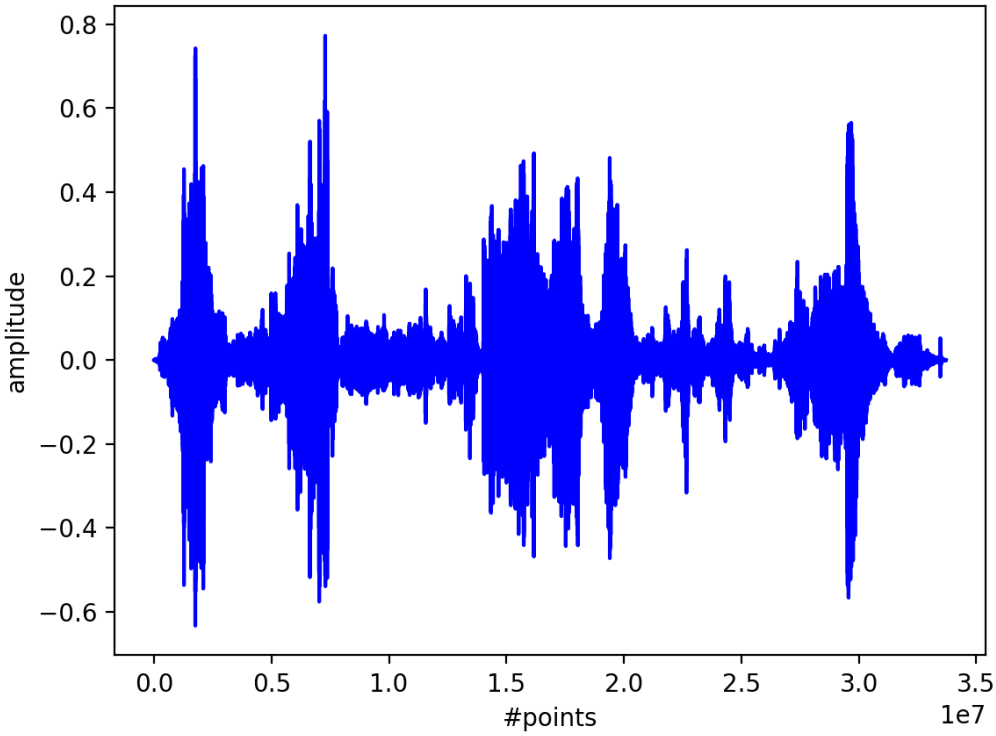

Figure 1: Line graph of the amplitudes

In the figure above are represented around 33.7 millions amplitudes and the high number of points is the main reason that this graph does not really look like a like graph.

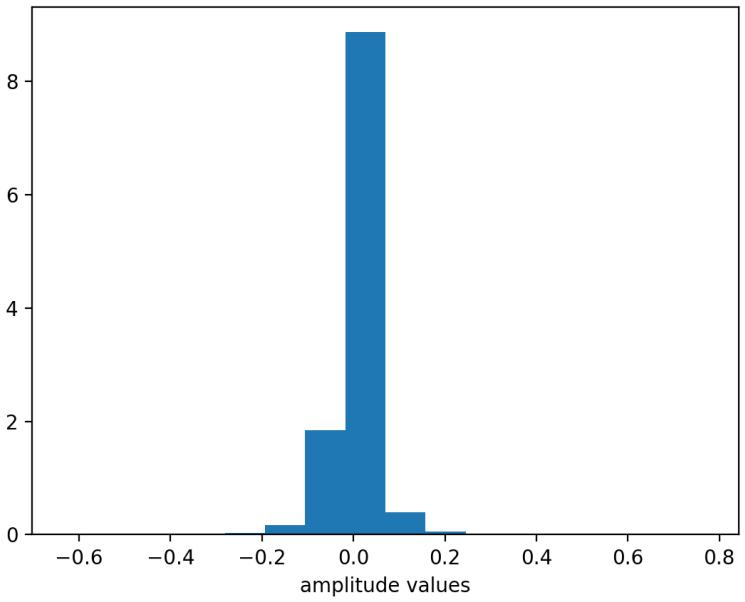

Figure 2: Histogram of the amplitudes

This form of the histogram is due to the large number of amplitudes with values very close to 0.

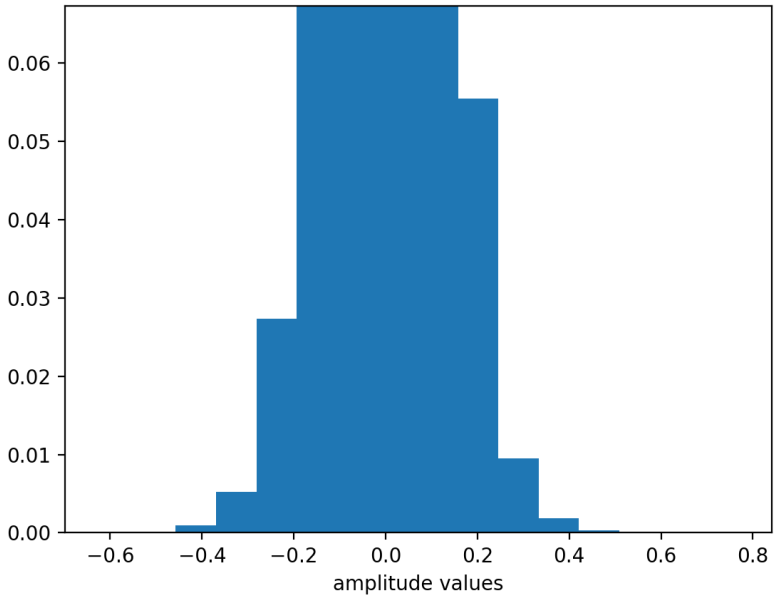

Figure 3: Histogram of the amplitudes (close view)

In order to give better view of the histogram I had to zoom in on the values far from 0.

1.2 Q1b

The splitting method:

```
# Split data into training, validation and test sets
def split_data(data):
    p = math.floor(len(data) * 0.15)     # 15% (validation set / testing set)
    r = len(data) - 2 * p                # 70% (training set)

    training_data = (data[:r])           # training_data
    validation_data = (data[r:r+p])      # validation_data
    testing_data = (data[r+p:r+p+p])     # testing_data

    X_shuffle_train = training_data[:,:-1]
    y_shuffle_train = training_data[:,-1]
    X_shuffle_val = validation_data[:,:-1]
    y_shuffle_val = validation_data[:,-1]
    X_shuffle_test = testing_data[:,:-1]
    y_shuffle_test = testing_data[:,-1]

    return X_shuffle_train, X_shuffle_val, X_shuffle_test,
                    y_shuffle_train, y_shuffle_val, y_shuffle_test
```

2 Question 2

2.1 Q2a

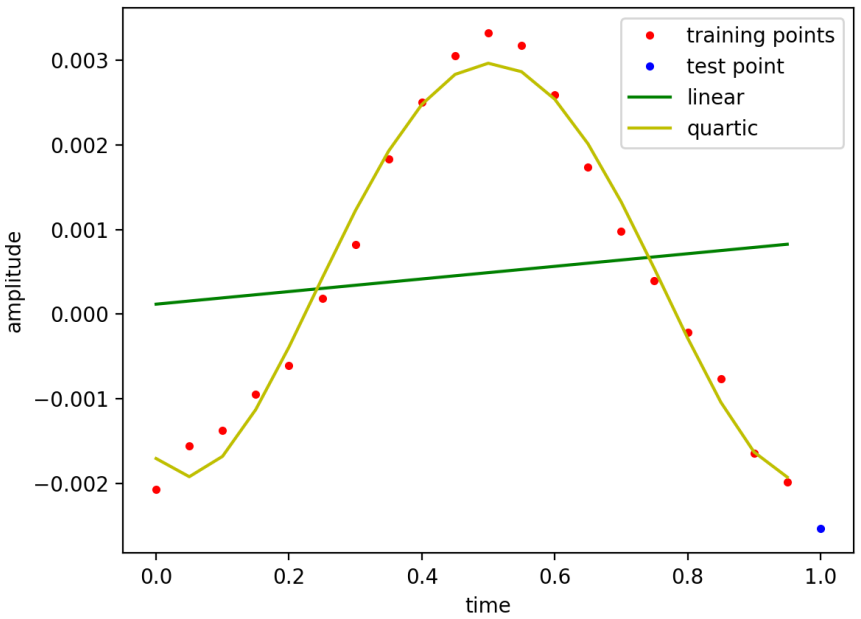

Figure 4: 20 training points, a test point, a straight line fit, and a quartic fit

Method **fit_curve** which plots the figure above (also includes the plotting for Q2b):

```
# Fit curves
def fit_curve(X, yy):
    X = X.reshape(20,1)
    yy = yy.reshape(1,1)
    D = X.shape[1]                                                    # N = 1, D = 20
    t = np.arange(start=0, stop=1, step=0.05).reshape(-1, D) # (N,D) - (20,1)

    print ("Expected value: ", yy[0][0])

    print ("USING ALL DATA")
    plt.clf()
    fit_and_plot(phi_fn=phi_linear, type='linear', X=t, yy=X, last_point=(1,
    yy[0]), draw_type='g-', D=D)
    fit_and_plot(phi_fn=phi_quartic, type='quartic', X=t, yy=X, last_point=(1,
    yy[0]), draw_type='y-', D=D)
    plt.ylabel("amplitude")
    plt.xlabel("time")
    plt.show()

    print ("USING JUST THE LAST TWO POINTS")                  # for Q2b
    plt.clf()
    fit_and_plot(phi_fn=phi_linear, type='linear', X=t[-2:], yy=X[-2:],
    last_point=(1, yy[0]), draw_type='g-', D=D)
    fit_and_plot(phi_fn=phi_quartic, type='quartic', X=t[-2:], yy=X[-2:],
    last_point=(1, yy[0]), draw_type='y-', D=D)
    plt.ylabel("amplitude")
    plt.xlabel("time")
    plt.show()

# Plot curves and compute estimation value for 21st point
def fit_and_plot(phi_fn, type, X, yy, last_point, draw_type, D):
    w_fit = np.linalg.lstsq(phi_fn(X), yy, rcond=0)[0]    # (D+1,)
    X_grid = np.arange(start=0, stop=1, step=0.05).reshape(-1, D)    # (N, 1)
    f_grid = np.dot(phi_fn(X_grid), w_fit)  # (N,)

    est_value = last_point[0] * w_fit[1] + w_fit[0]
    print ("Estimated value for", type, ":", est_value[0])

    plt.plot(X, yy, 'r.')
    plt.plot(last_point[0], last_point[1], 'b.')
    plt.plot(X_grid, f_grid, draw_type, label=type)
    plt.legend()

# Linear function
def phi_linear(X_in):
    return np.insert(X_in, 0, 1, axis=-1)

# Quartic function
def phi_quartic(X_in):
    return np.concatenate([np.ones(X_in.shape), X_in, X_in**2, X_in**3, X_in
    **4], axis=-1)
```

2.2 Q2b

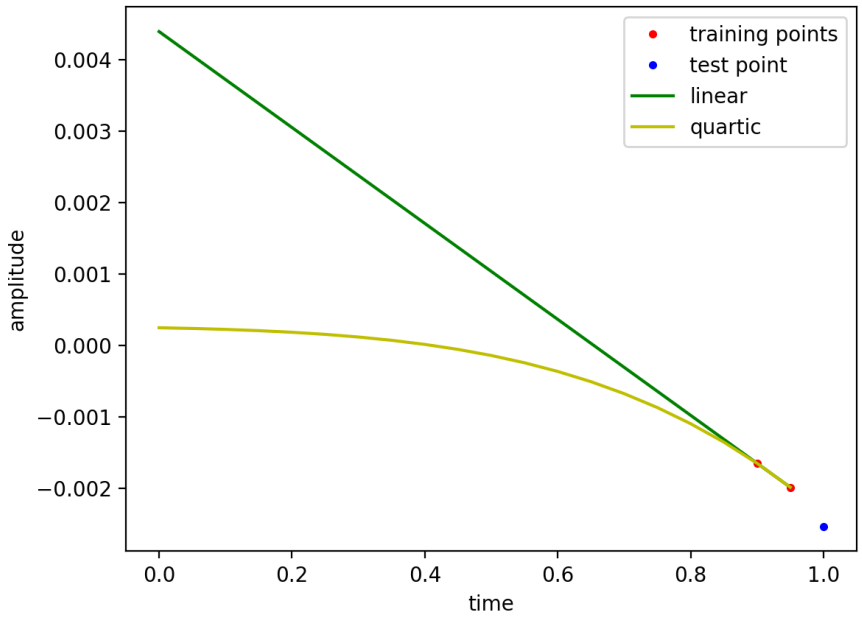

Figure 5: Last 2 training points, a test point, a straight line fit, and a quartic fit

Expected value: -0.0025

Estimated value for linear : 0.0008
Estimated value for quartic : -0.0116

Using all 20 point we can see that quartic model give a better estimation than the linear model for the 21st point.

Estimated value for linear : -0.0023
Estimated value for quartic : 7.9113e-05

Plotting just the last 2 points we are observing that linear model brings a good estimation for the 21st point while the quartic model is way off. This is happening because of overfitting (the grade of the polynomial is greater than the number of the points).

For a small context, the straight line provides the appropriate solution with lower number of variables. This avoids complexity of calculations. As the degree of polynomial increases, the chances of getting multiple solutions increase and so is the complexity of the calculations. This will also result in overfitting of the training data. To avoid complexity, for smaller contexts we chose the lower degree polynomials.

2.3 Q2c

If we look at one row from the training set, which is 1xN (where N = 20, it contains 20 values). In order to not get into overfitting and also to give the best prediction we think that the order of the polynomial should be less or equal than N.

3 Question 3

$$\mathbf{w} = (\Phi^{\mathrm{T}}\Phi)^{-1}(\Phi^{\mathrm{T}}\mathbf{x})$$

3.1 Q3a

We have the model's prediction equation for t=1:

$$f(t=1) = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(t=1)$$

$$f(t=1) = \mathbf{v}^{\mathrm{T}}\mathbf{x}$$

We need to resolve the linear equation:

$$\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(t=1) = \mathbf{v}^{\mathrm{T}}\mathbf{x}$$

and get $\mathbf{v}$.

Since $\mathbf{v}^{\mathrm{T}}\mathbf{x}$ is a scalar, it is equal with its transpose:

$$\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(t=1) = \mathbf{x}^{\mathrm{T}}\mathbf{v}$$

$$[(\Phi^{\mathrm{T}}\Phi)^{-1}(\Phi^{\mathrm{T}}\mathbf{x})]^{\mathrm{T}}\phi(t=1) = \mathbf{x}^{\mathrm{T}}\mathbf{v}$$

$$(\Phi^{\mathrm{T}}\mathbf{x})^{\mathrm{T}}[(\Phi^{\mathrm{T}}\Phi)^{-1}]^{\mathrm{T}}\phi(t=1) = \mathbf{x}^{\mathrm{T}}\mathbf{v}$$

$$\mathbf{x}^{\mathrm{T}}\Phi(\Phi^{\mathrm{T}}\Phi)^{-\mathrm{T}}\phi(t=1) = \mathbf{x}^{\mathrm{T}}\mathbf{v}$$

$$\mathbf{v} = \Phi(\Phi^{\mathrm{T}}\Phi)^{-\mathrm{T}}\phi(t=1)$$

3.2 Q3bi

```
'''
This code creates a design matrix phi(c,k) which constructs the matrix in the
    form of phi = [1 t t**2...]
'''
def Phi(c,k):
  x = np.arange(0, 1, 0.05)
  phi_values = x[-c:]
  phi_matrix = phi_values.reshape(1, c)
  for i in range(k):
    if i > 1:
      phi_values = (phi_values ** i).reshape(1, c)
      phi_matrix = np.concatenate([phi_matrix, phi_values], axis=0)
  phi_matrix = np.matrix.transpose(phi_matrix)
  phi = np.concatenate([np.ones((phi_matrix.shape[0], 1)), phi_matrix], axis
    =1)

  return phi
```

3.3 Q3bii

```
# Get v vector
def make_vv(c, k):
  for i in range(1, c+1, 1):
    for j in range(1, k+1, 1):
      x = Phi(c=i, k=j)
      x_transpose = np.matrix.transpose(x)
      v = np.linalg.pinv(x_transpose.dot(x)).dot(x_transpose)

  return v
```

3.4 Q3biii

```
columns = 21
rows = len(amp_data_keys) // columns
array_gen = np.reshape(amp_data_keys[:(rows * columns)], (rows, columns))

'''
Two vectors v make the same predictions at t=1 like the linear and quadratic
    functions from Question 2
'''
def vv_least_squares(c,k):
  v = make_vv(c, k)
  X_shuf_train, Y_shuf_train, X_shuf_valid, Y_shuf_valid, X_shuf_test, Y_shuf_test
    = data_split(array_gen)
  y_initial = X_shuf_train[0]
  quartic_vv = np.dot(v, y_initial[-c:])
  quartic_lstsq = np.linalg.lstsq(Phi(c=c, k=k), y_initial[-c:], rcond=0)[0]
  print ("Weight matrix with least squares:", quartic_lstsq)
  print ("Weight matrix with with vv:", quartic_vv)


vv_least_squares(c=1, k=4)
```

Result for linear:
Weight matrix with least squares:
[-3.20815539e-05 -3.04774762e-05]
Weight matrix with with vv:
[-3.20815539e-05 -3.04774762e-05]

Result for quartic:
Weight matrix with least squares:
[-1.87375781e-05 -1.78006992e-05 -1.69106643e-05 -1.37738417e-05]
Weight matrix with with vv:
[-1.87375781e-05 -1.78006992e-05 -1.69106643e-05 -1.37738417e-05]

3.5 Q3ci

```
# Setting of K and C for the smallest square error on training set
def min_leastsquare_inv(c,k):
  X_shuf_train, Y_shuf_train, X_shuf_valid, Y_shuf_valid, X_shuf_test,
   Y_shuf_test = data_split(array_gen)
  y_expected = Y_shuf_train[1]
  y_predicted = 0
  min_square_error = 1000 # A random value for error comparison to get the
   least value. Can be any value. Should be large
  y_initial = X_shuf_train[1]
  c3=0
  k3=0
  for i in range(1, c+1, 1):
    for j in range(1, k+1, 1):
      v = make_vv(c=i, k=j)
      w = np.dot(v, y_initial[-i:])
      y_predicted = np.dot([np.ones((w.shape[0]))], w)
      min_error = y_expected - y_predicted
      if min_error < min_squareerror:
        min_square_error = min_error
        c3 = i
        k3 = j
  print ("Minimum square error with v:", min_squareerror)
    print ("Value for k:", k3)
    print ("Value for c:", c3)


min_leastsquare_inv(c=1, k=10)
```

Result:
Minimum square error with v: [-0.0001014]
Value for k: 6
Value for c: 1

3.6 Q3cii

```
# Mean square error on training, validation and test sets
def min_leastsquare_data(c,k):
  X_shuf_train, Y_shuf_train, X_shuf_valid, Y_shuf_valid, X_shuf_test, Y_shuf_test
   = data_split(array_gen)
  y_predicted = 0
  mean_square_error = 0
  min_error = 0
  for i in range(len(Y_shuf_test)):
    y_expected = Y_shuf_test[i]
    y_initial = X_shuf_test[i]
    for j in range(1, c+1, 1):
      y = y_initial[-j:]
      for r in range(1, k+1, 1):
        v = make_vv(c=j, k=r)
        w = np.dot(v, y)
        y_predicted = np.dot([np.ones((w.shape[0]))], w)
        min_error += y_expected - y_predicted
  mean_square_error = min_error / len(Y_shuf_test)

  print ("Mean square error with inverse operation:", mean_square_error)


min_leastsquare_data(c=1, k=4)
```

Training data for CxK = 1x4:
Mean square error with inverse operation: 4.952014950010723e-10

Validation data for CxK = 1x4:
Mean square error with inverse operation: -6.6993145829630385e-09

Test data for CxK = 1x4:
Mean square error with inverse operation: 6.721897075015231e-10

4 Question 4

4.0.1 Q4a

```
# Compute mean square error
def min_leastsquare(c,k):
  X_shuf_train, Y_shuf_train, X_shuf_valid, Y_shuf_valid, X_shuf_test, Y_shuf_test
   = data_split(array_gen)
  y_predicted = 0
  min_square_error = 1000
  c4 = 0
  k4 = 0
  for i in range(len(Y_shuf_valid)):
    y_initial = X_shuf_valid[i]
    y_expected = Y_shuf_valid[i]
    for j in range(1, c+1, 1):
      y = y_initial[-j:]
      for r in range(1, k+1, 1):
        x = Phi(c=j, k=r)
        w = np.linalg.lstsq(x, y, rcond=None)[0]
        y_predicted = np.matrix.transpose(np.dot([np.ones((w.shape[0]))], w))
        min_error = y_expected-y_predicted
        if min_error < min_square_error:
          min_squareerror = min_error
          c4 = j
          k4 = r
    print ("y_expected:", y_expected)
    print ("y-predicted:", y_predicted)
    print ("Value of C:", c4,", value of K:", k4)
    print ("Least square error:", min_squareerror "at value of C:", c4," and
   value of K:", k4)
    return (c4, k4)


min_leastsquare(c=10, k=20)
```

Result on validation data:
y_expected: -0.01226806640625
y-predicted: [-2.95059646]
Value of C: 5, value of K: 7
Least square error: [-19.53224823] at value of C: 5 and value of K: 7

4.1 Q4b

```
# Compare validation error results
def test_data_meanerror(c,k):
  X_shuf_train, Y_shuf_train, X_shuf_valid, Y_shuf_valid, X_shuf_test, Y_shuf_test
   = data_split(array_gen)
  min_error = 0
  mean_square_error = 0
  for i in range(len(Y_shuf_test)):
    y_initial = X_shuf_test[i]
    y_expected = Y_shuf_test[i]
    x = Phi(c=c, k=k)
    y = y_initial[-c:]
    w = np.linalg.lstsq(x, y, rcond=None)[0]
    y_predicted = np.matrix.transpose(np.dot([np.ones((w.shape[0]))], w))
    min_error += y_expected - y_predicted
  mean_square_error = min_error / len(Y_shuf_test)
  print("Mean square error for the best validation fit on test data with least
    squares:", mean_square_error)
  return y_predicted


test_data_meanerror()
```

Mean square error for the best validation fit on test data with least squares: [-0.00068315]

We can clearly see that the validation error from question 3 (mean square error: -6.6993145829630385e-09 for polynomial of grade 4) is better than the validation error using the best predictor (mean square error: -0.00068315) over the previous 20 amplitudes available.

4.2 Q4c

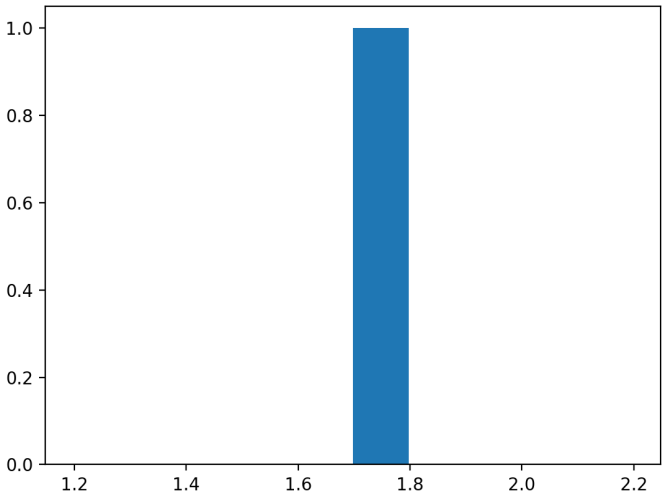

Figure 6: Validation data residual histogram