School of Informatics

Machine Learning & Pattern Recognition
Assignment 1

David Strömbäck (s1874170)
Lucas Pompe (s1848701)
April 2020

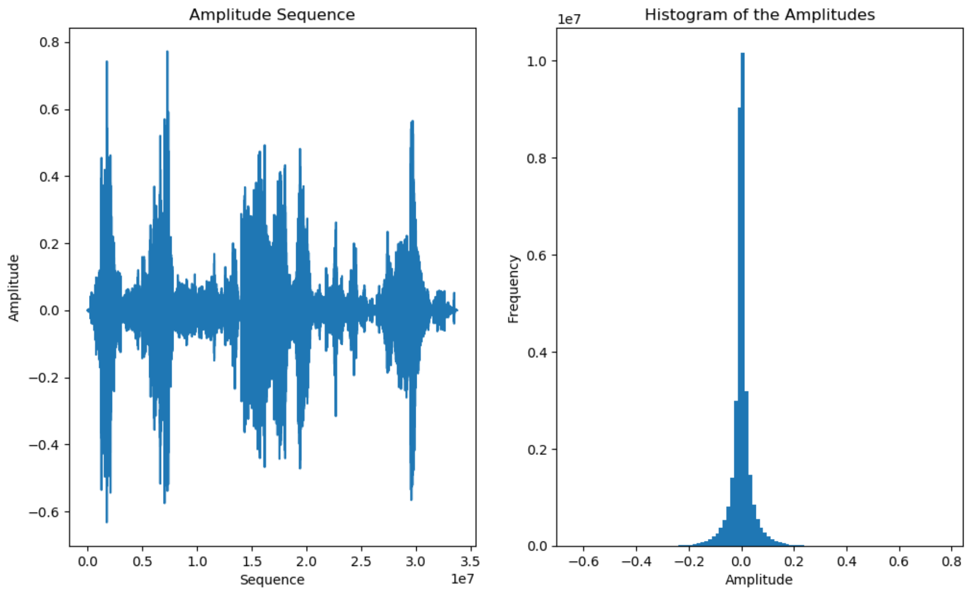

Figure 1: Amplitude plot of the audio sequence and a histogram of the amplitudes.

1: Getting started

1a)

```python
import scipy.io as sio
import matplotlib.pyplot as plt
import numpy as np

amp = sio.loadmat('amp_data.mat')
amp_data = amp['amp_data'].squeeze()

plt.figure(figsize=(12, 7))
plt.subplot(121)
plt.xlabel('Sequence')
plt.ylabel('Amplitude')
plt.title('Amplitude Sequence')
plt.plot(amp_data)
plt.subplot(122)
plt.xlabel('Amplitude')
plt.ylabel('Frequency')
plt.title('Histogram of the Amplitudes')
plt.hist(amp_data, bins=100)
plt.show()
```

Two things seem to stand out for this data in particular:

- Most of the recorded amplitudes are zero, and the distribution of data points seem to decay exponentially as the distance from zero is increased.
- As visible in the sequence plot: deviations from the mean seem to happen in a correlated fashion over time. Meaning that when the amplitude starts to deviate over time, it increases the probability of the amplitude in the next time-steps to deviate as well, causing

clusters of deviation over time.

1b)

```python
'''Discard some elements to ensure the 'amp_data' vector length is a multiple of
    'lag'. Wrap the amp_data vector into a Cx'lag' matrix, where each row contains
     'lag' amplitudes that were adjacent in the original sequence.'''
lag = 21
reshaped_data = np.reshape(amp_data[len(amp_data) % lag:], (-1, lag))
X = reshaped_data[:, :-1]
y = reshaped_data[:, -1]

def gen_crossval_set(X, y, train_frac=.7, val_frac=.15, test_frac=.15):
    np.random.seed(1)
    indices = np.random.permutation(np.arange(len(X)))

    train_ind_start = 0
    train_ind_end = int(len(X) * train_frac)

    val_ind_start = train_ind_end
    val_ind_end = val_ind_start + int(len(X) * val_frac)

    test_ind_start = val_ind_end
    test_ind_end = test_ind_start + int(len(X) * test_frac)

    train_is = indices[train_ind_start:train_ind_end]
    val_is = indices[val_ind_start:val_ind_end]
    test_is = indices[test_ind_start:test_ind_end]

    return {
        "train": (X[train_is], y[train_is]),
        "val": (X[val_is], y[val_is]),
        "test": (X[test_is], y[test_is])
    }

data_dict = gen_crossval_set(X, y)

X_shuf_train, y_shuf_train = data_dict['train']
X_shuf_val, y_shuf_val = data_dict['val']
X_shuf_test, y_shuf_test = data_dict['test']
```

2: Curve fitting a snippet of audio

2a)

```python
t = np.linspace(0, 1, lag-1, endpoint=False)
x_example = X_shuf_train[3, None].T
y_example = y_shuf_train[3]

phi_linear = lambda x: np.vstack([np.ones(len(x)), x]).T
phi_quartic = lambda x: np.vstack([np.ones(len(x)), x, x**2, x**3, x**4]).T

fitted_linear_weights = np.linalg.lstsq(phi_linear(t), x_example, rcond=None)[0]
fitted_quartic_weights = np.linalg.lstsq(phi_quartic(t), x_example, rcond=None)
    [0]

t_with_test_point = np.append(t, 1)
linear_fit = np.matmul(phi_linear(t_with_test_point), fitted_linear_weights)
```

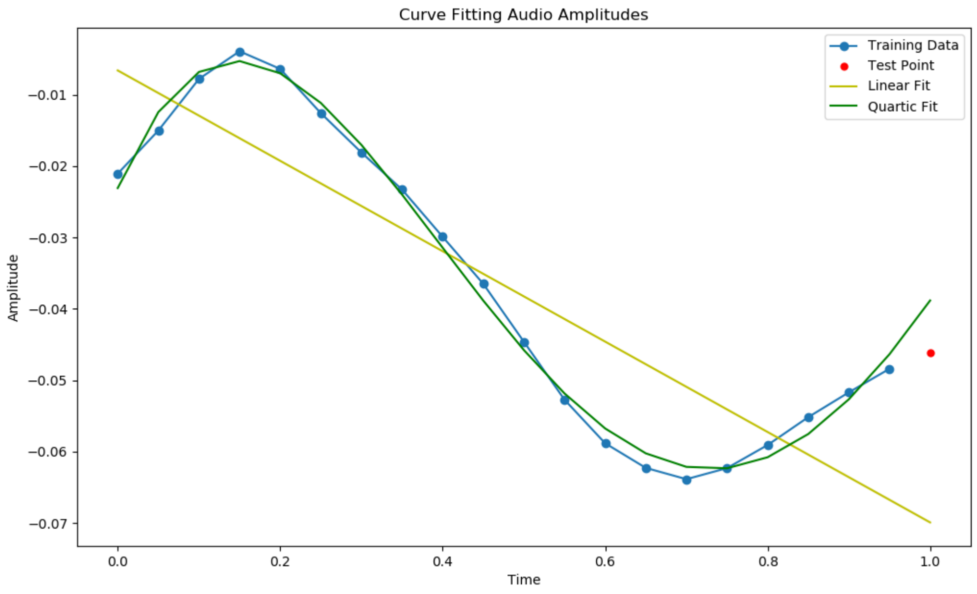

Figure 2: Linear and quartic fit using least squares on .

```
quartic_fit = np.matmul(phi_quartic(t_with_test_point), fitted_quartic_weights)
```

2b)

If we assume the true underlying function to be some nonlinear function, we can approach the smallest possible change using a linear function (of course, this is how derivatives work). So as we measure points that are closer together, their relationship will approach a linear function. We can't actually estimate the gradient between $t = \frac{20}{20}$ and the test point, because the test point is unknown. Hence, we estimate it by estimating the gradient (so fitting a linear function) using the last two known points. Adding more points increases the loss, since these points don't originate from a linear function, and therefore (on average) don't provide good information for estimating the gradient near the test point.
The quartic fit performs worse with less data since fewer data points can't encode enough information to describe the more complex curvature of a quartic function.

2c)

The more naive hypothesis here would perhaps be to predict that a quartic fit would perform better (with context length equal to 20) because it seems to fit all the training data well. However, while the quartic fit might seem to fit the training data well, in a number of cases from the training set the quartic fit does not extrapolate well to the 21st point. In figure 3 for example: the quartic fit overshoots the 21st point, undershooting happens in some cases as well, suggesting that the quartic model might be overfitting the training data. From inspecting a couple of different sequences, it seems that the simple linear model (with context length equal to 2) would perhaps perform the best. The linear model is less prone to over- and undershoot because of the reasons mentioned in our answer to question 2b.

3: Choosing a polynomial predictor based on performance

3a)

$$\begin{aligned} f(t=1) &= [(\Phi^T\Phi)^{-1}(\Phi^T \boldsymbol{x})]^T \phi(t=1) && (1)\\ &= \boldsymbol{x}^T \Phi(\Phi^T\Phi)^{-T}\phi(t=1) && (2)\\ &= [\Phi(\Phi^T\Phi)^{-T}\phi(t=1)]^T \boldsymbol{x} && (3)\\ &= [\Phi(\Phi^T\Phi)^{-1}\phi(t=1)]^T \boldsymbol{x} && (4)\end{aligned}$$

Let $\boldsymbol{v} = \Phi(\Phi^T\Phi)^{-1}\phi(t=1)$, which is a $C \times 1$ matrix. Since $\boldsymbol{x}^T$ is a $1 \times C$ matrix we can rearrange Equation 2 to get Equation 3.

3b)

i)

```python
def phi(C, K):
    t = np.linspace(0, 1, 20, endpoint=False)[-C:]
    _, yv = np.meshgrid(np.arange(0, K), t)
    basis_powers = np.arange(0, K)
    phi = np.power(yv, basis_powers)
    return phi
```

ii)

```python
def make_vv(C, K):
    p = phi(C, K)
    v = np.matmul(np.matmul(p, np.linalg.inv(np.matmul(p.T,p))), np.ones((K, 1)))
    return v
```

iii)

```python
linear_v = make_vv(20, 2)
quartic_v = make_vv(20, 5)

lin = np.matmul(linear_v.T, x_example)
qua = np.matmul(quartic_v.T, x_example)

plt.figure(figsize=(12, 7))
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('Curve Fitting Audio Amplitudes')
plt.plot(t, x_example, '-o', label='Training Data')
plt.plot(1, y_example, 'r.', label='Test Point', markersize=10)
plt.plot(1, lin, 'm.', label='Linear Prediction using v', markersize=20)
plt.plot(1, qua, 'c.', label='Quartic Prediction using v', markersize=20)
plt.plot(t_with_test_point, linear_fit, 'y', label='Linear Fit')
plt.plot(t_with_test_point, quartic_fit, 'g', label='Quartic Fit')
plt.legend()
plt.show()
```

3c)

i)

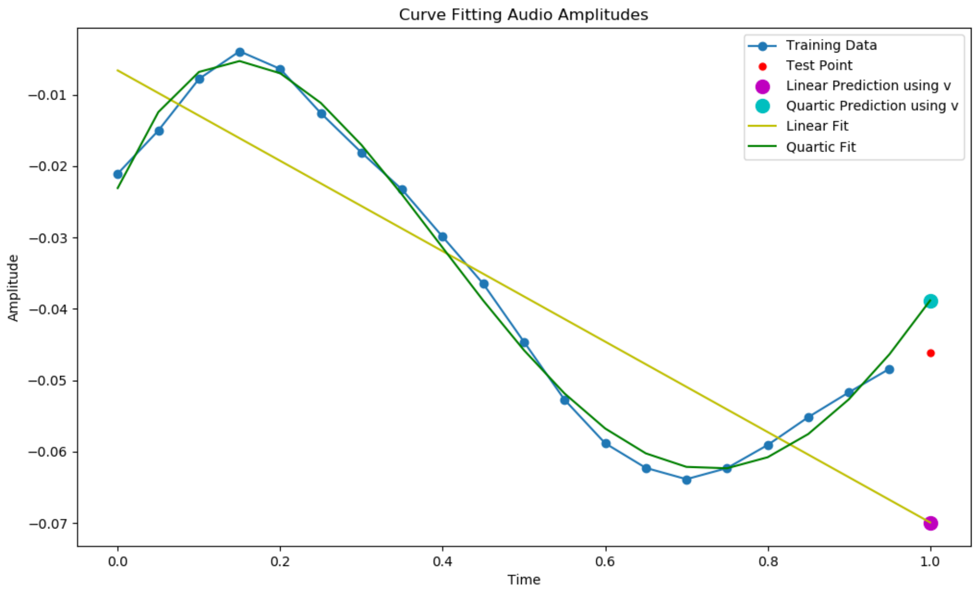

Figure 3: Plot demonstrating that using v derived in 3a to predict the next amplitude, achieves the same results as using a linear least squares fit.

```python
c_max = 20
k_max = 10

errors = np.zeros((c_max, k_max))
for c in range(1, c_max+1):
    print('C:', c)
    for k in range(1, k_max+1):
        v = make_vv(c, k)
        pred = np.matmul(v.T, X_shuf_train[:, -c:].T)
        sq_error = (pred-y_shuf_train)**2
        errors[c-1, k-1] = np.mean(sq_error)

ax = sns.heatmap(errors, linewidth=0.5, xticklabels=range(1, k_max+1),
    yticklabels=range(1, c_max+1), vmin=0, vmax=1e-3,
                    cbar_kws={'label': 'Average Mean Square Error'})
plt.title('Grid Search for C and K Parameters on Training Set')
plt.ylabel('C')
plt.xlabel('K')
plt.show()

print(np.min(errors))
min_indices = np.unravel_index(np.argmin(errors), errors.shape)
print('C:', min_indices[0]+1, 'K:', min_indices[1]+1)
```

We found that using $C = 2$ and $K = 2$ as parameters achieves the lowest mean squared error on the training set.

ii)

Train Squared Error: 1.3318e-05
Test Squared Error: 1.3760e-05

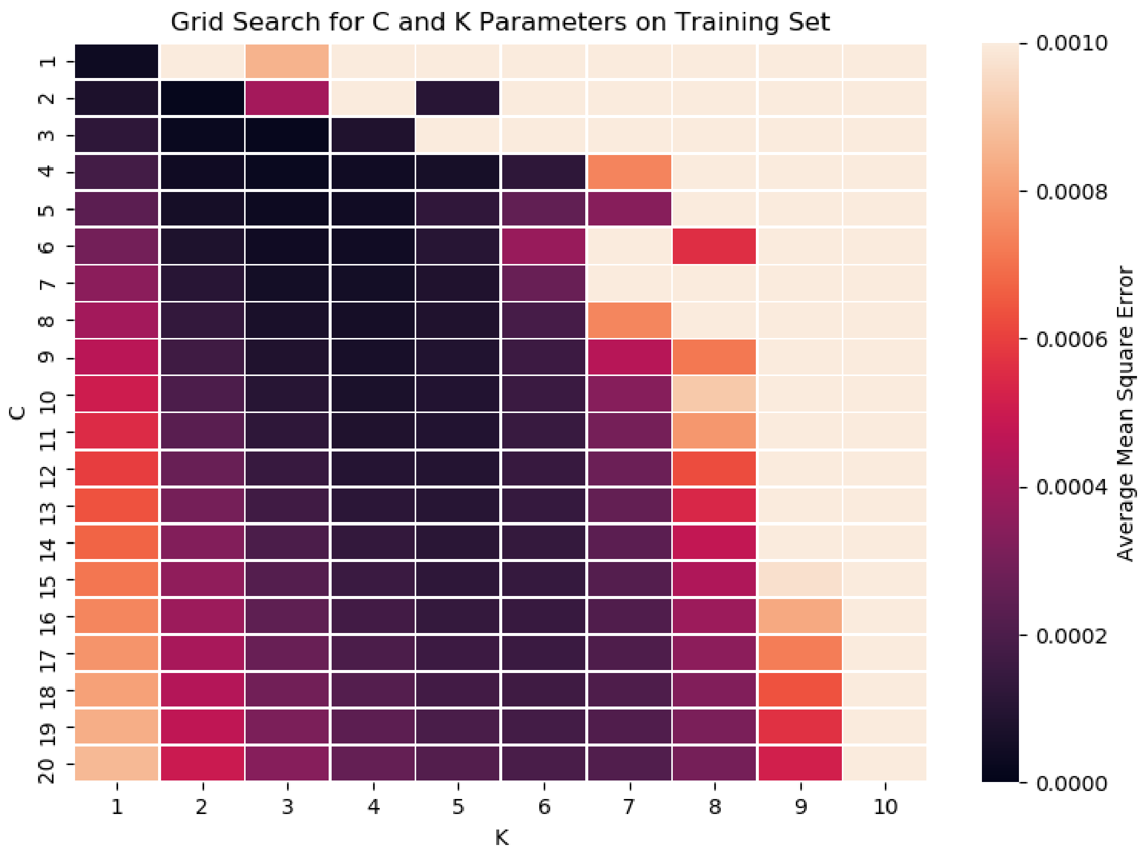

Figure 4: Heat-map for the training loss per parameter combination of C and K.

Validation Squared Error: 1.3508e-05

```python
best_v = make_vv(2, 2)
pred = np.matmul(best_v.T, X_shuf_train[:, -2:].T)
sq_error = (pred - y_shuf_train) ** 2
print('Train Squared Error', np.mean(sq_error))
pred = np.matmul(best_v.T, X_shuf_val[:, -2:].T)
sq_error = (pred - y_shuf_val) ** 2
print('Validation Squared Error', np.mean(sq_error))
pred = np.matmul(best_v.T, X_shuf_test[:, -2:].T)
sq_error = (pred - y_shuf_test) ** 2
print('Test Squared Error', np.mean(sq_error))
```

4: Fitting linear predictors across many snippets

4a)

On both the training and validation set, using the maximum context length, $C = 20$, to fit a linear least squares model resulted in the smallest mean square error. The mean squared error was 7.7502e-06 on the training set and 7.9622e-06 on the validation set. The shorter the context length C used to fit the linear least squares model, the higher the mean squared error was on both the training and validation set. A reason as to why models fit using a longer context length performed better than models fit using shorter context lengths, is that the longer the context length, the more parameters the model has to try to get a closer fit to the training data. One might have expected that increasing the context length could have led to overfitting, but the results on the validation set suggest that this did not occur. Looking at the chosen weights for the fitted model using $C = 20$, it is noteworthy that the magnitude of the weights are smaller the further away they are in time from the time-step we are predicting, perhaps suggesting that the latter points are more important in making good predictions, which is also what one would expect.

```python
def get_linear_lstsq_for_diff_c(x, y):
    max_c = 20
    errors_c = np.zeros(max_c)
    for c in range(1, max_c+1):
        fitted_linear_weights = np.linalg.lstsq(X_shuf_train[:, -c:],
    y_shuf_train, rcond=None)[0]
        pred = np.matmul(fitted_linear_weights, x[:, -c:].T)
        sq_error = (pred - y) ** 2
        errors_c[c-1] = np.mean(sq_error)
    return errors_c

print(get_linear_lstsq_for_diff_c(X_shuf_train, y_shuf_train))
print(get_linear_lstsq_for_diff_c(X_shuf_val, y_shuf_val))
```

4b)

The standard linear least squares model fit using context length, $C = 20$, achieved a mean squared error on the test set of 8.0267e-06, outperforming the model fit in Question 3, which achieved a mean squared error of 1.3760e-05 on the test set.

```python
c = 20
fitted_linear_weights = np.linalg.lstsq(X_shuf_train[:, -c:], y_shuf_train, rcond
    =None)[0]
```

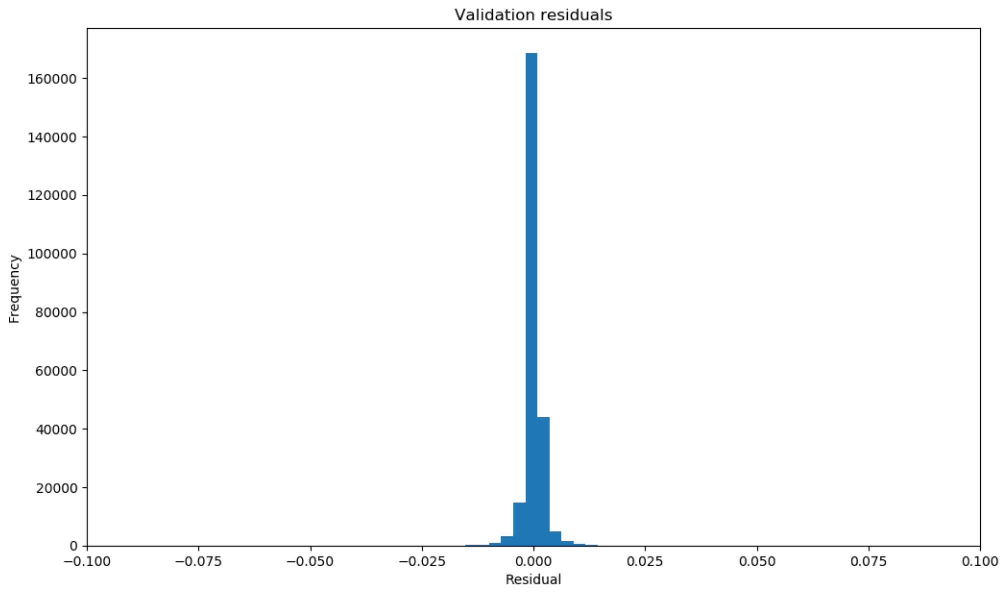

Figure 5: Residuals on the validation set

```
pred = np.matmul(fitted_linear_weights, X_shuf_test[:, -c:].T)
mean_sq_error = np.mean((pred - y_shuf_test) ** 2)
print('Test Mean Squared Error using Standard Linear lstsq', mean_sq_error)

best_v = make_vv(2, 2)
pred_v = np.matmul(best_v.T, X_shuf_test[:, -2:].T)
mean_sq_error_v = np.mean((pred_v - y_shuf_test) ** 2)
print('Test Mean Squared Error with C=2 and K=2', mean_sq_error_v)
```

4c)

```
c = 20
fitted_linear_weights = np.linalg.lstsq(X_shuf_train[:, -c:], y_shuf_train, rcond
    =None)[0]
pred = np.matmul(fitted_linear_weights, X_shuf_val[:, -c:].T)

plt.figure(figsize=(12, 7))
plt.xlim(-.10, .10)
plt.title('Validation residuals')
plt.xlabel('Residual')
plt.ylabel('Frequency')
plt.hist(pred - y_shuf_val, bins=100)
plt.show()
plt.legend()
plt.show()
```

Figure 5 shows the residuals on the validation set. Like the distribution of the amplitudes, the distribution of the residuals is centered around 0, with most values very close to 0. Having the residuals centred around 0, with a mean of approximately 0 is what is expected when fitting a linear least squares model.

5

- Currently we only use every 21$^{\text{st}}$ point for prediction, and even though we don't really have a data shortage in this case, more complicated models require vast amounts of data. One straightforward way of increasing our amount of training data is lagging the data. This technique is very popular in time series analysis (TSA), and will greatly enhance the amount of data we can work with. Instead of using every n-th data point as target and it's C context points as features, data lagging constructs one giant matrix that contains the C context points for every single data point in the time series. (For Python example see: statsmodels.tsa.tsatools.lagmat). Additionally, Fourier transformations are a popular method to decompose sound into it's individual components.

- In Question 1a we mentioned that amplitude deviation happens in a clustered fashion over time. We might be able to exploit this property using some popular TSA techniques like auto-regression ((V)AR) or regime-shift. AR is the ordinary linear regression we've been doing so far in the assignment, scaled up to the lagged data described above. Vector auto-regression uses additional features to predict the next timestep in the series. This would obviously require us to obtain additional feature data.
  Regime shift algorithms attempt to classify different clusters (regimes) of activity in the time series. The idea being that if we can classify these regimes, maybe we will find a repeating pattern of regimes we can use to predict upcoming regimes.