Exam in Mathematical Modelling & Analysis (MAD), 2020-21

Exam number: 23

January 22, 2021

Contents

Statistics

Question 1 (Maximum Likelihood Estimation)

Fixing the parameter σ, we can express the likelihood L of observing the dataset $x_1, \ldots, x_N$ of N i.i.d. samples from X as a the joint distribution of all of the N i.i.d. samples, conditioning on μ:

$$L = p(x_1, \ldots, x_N|\mu) = \prod_{n=1}^{N} p(x_n|\mu) = \prod_{n=1}^{N} f(x_n; \mu)$$

where the PDF is now considered a function of both x_n and μ.

We want to find the optimal $\hat{\mu}$ that maximizes the likelihood L:

$$\hat{\mu} = \underset{\mu}{\operatorname{argmax}}\, L$$

Taking the log makes the expression more managable—without changing the estimate—and we get

$$\begin{aligned}
\log L &= \log\left(\prod_{n=1}^{N} f(x_n; \mu)\right) \\
&= \sum_{n=}^{N} \log(f(x_n; \mu)) \\
&= \sum_{n=1}^{N} \left[\log\left(\frac{1}{\sigma\sqrt{2\pi}}\right) + \log\left(\frac{1}{x_n}\right) + \log\left(\exp\left[-\frac{1}{2}\left(\frac{\log(x) - \mu}{\sigma}\right)^2\right]\right)\right] \\
&= \sum_{n=1}^{N} \left[\log\left(\frac{1}{\sigma\sqrt{2\pi}}\right) + \log\left(\frac{1}{x_n}\right) - \frac{1}{2}\left(\frac{\log(x) - \mu}{\sigma}\right)^2\right]
\end{aligned}$$

To find the maximum likelihood estimate $\hat{\mu}$ for the parameter μ we optimize L by solving $\frac{\partial \log L}{\partial \mu} = 0$ with respect to μ. When taking derivatives we can ignore terms that do not involve μ, and so we first get

$$\begin{aligned}
\frac{\partial \log L}{\partial \mu} &= \frac{\partial}{\partial \mu}\left[\sum_{n=1}^{N} -\frac{1}{2}\left(\frac{\log(x) - \mu}{\sigma}\right)^2\right] \\
&= \sum_{n=1}^{N} -\left(\frac{\log(x) - \mu}{\sigma}\right) \times \left[\frac{\log(x) - \mu}{\sigma}\frac{\partial}{\partial \mu}\right] \qquad (\textit{Chain Rule}) \\
&= \sum_{n=1}^{N} -\left(\frac{\log(x) - \mu}{\sigma}\right) \times (-\frac{1}{\sigma}) \\
&= \frac{1}{\sigma^2}\sum_{n=1}^{N} \log(x_n) - \mu
\end{aligned}$$

Solving $\frac{\partial \log L}{\partial \mu} = 0$ with respect to μ:

$$0 = \frac{1}{\sigma^2}\sum_{n=1}^{N}\log(x_n) - \mu = \sum_{n=1}^{N}\log(x_n) - \mu = \left[\sum_{n=1}^{N}\log(x_n)\right] - N\mu$$

$$\Updownarrow$$

$$N\mu = \sum_{n=1}^{N}\log(x_n)$$

$$\Updownarrow$$

$$\mu = \frac{1}{N}\sum_{n=1}^{N}\log(x_n) = \frac{1}{N}\log(x_1 \cdot x_2 \cdot \ldots \cdot x_N) = \log\Big((x_1 \cdot x_2 \cdot \ldots \cdot x_N)^{1/N}\Big)$$
$$= log(\sqrt[N]{x_1 \cdot x_2 \cdot \ldots \cdot x_N})$$

The above is a neccesary condition for an optimal solution. We have further

$$\frac{\partial^2 \log L}{\partial \mu^2} = -\frac{N}{\sigma^2}$$

which is everywhere negative. Thus, by the second derative test

$$\widehat{\mu} = log(\sqrt[N]{x_1 \cdot x_2 \cdot \ldots \cdot x_N})$$

is a global maximum. QED.

Question 2 (Hypothesis Testing)

There already is formulated a model ($X \sim \mathcal{N}(\mu, \sigma^2 = 0.1)$) for the data as well as the *null hypothesis* $H_0 : \mu \geq 8.5$ to be tested against the *alternative hypothesis* $H_A : \mu < 8.5$ (i.e. we are doing a *left-sided* hypothesis test). The significance level $\alpha = 5\%$ has also already been chosen. We are left to specify the relevant *test statistics*, compute the *rejection region*, and reject appropriately.

- *Specifying test statistics:* Since we know the variance $\sigma^2 = 0.1$ and since the data set $x_1, \ldots, x_n$ are i.i.d. samples from $X \sim \mathcal{N}(\mu, \sigma^2 = 0.1)$, we can choose the Normal distributed r.v.

$$Z = \sqrt{n}\left(\frac{\bar{x} - \mu}{\sigma}\right) \sim \mathcal{N}(0, 1),$$

  where $\bar{x}$ is the sample mean and n is the number of samples, as our test statistics.

  Using the following line of code in Python

```python
import numpy as np
import scipy.stats

# known variance
sigma_squared = 1.0

# samples
X = np.array([8.2, 7.9, 8.7, 8.3, 8.5, 8.3, 8.8, 8.2, 8.7, 7.6, 8.4])
n = len(X)

# sample mean
X_mean = np.mean(X)

# z-test
mu = 8.5
alpha = 0.05
z = np.sqrt(n) * (X_mean - mu) / np.sqrt(sigma_squared)
```

  gives us the test statistics for the samples

$$z = \sqrt{n}\left(\frac{\bar{x} - \mu}{\sigma}\right) = \sqrt{11}\left(\frac{8.3273 - 8.5}{\sqrt{0.1}}\right) = -0.5729$$

- *Computing rejection region:* Using inverse lookup in the Normal distribution CDF at

$$P(Z \geq c) = \alpha$$

  in Python with

```
24  c = scipy.stats.norm.ppf(alpha)
```

gives us the critical region (i.e. the *rejection region*):

$$\mathcal{R} = (\infty; -1.6449]$$

- *Reject or not?* Clearly, $z = -0.5729$ is *not* inside the rejection region, so we do *not* reject the null hypothesis H_0.

Note, that alternatively we could have also used the *sample variance* and the Student-t distribution instead in a so-called *t-test* to get to a similar conclusion.

The complete source code can be found the in Appendix A. Below is the out from running the Python script

```
Known variance: 1.0
Samples: [8.2 7.9 8.7 8.3 8.5 8.3 8.8 8.2 8.7 7.6 8.4]
Number of samples: 11
Sample mean x: 8.3273
Significance Level alpha: 0.05
Test statistics for the samples z: -0.5729
Critical Value: -1.6449
```

Question 3 (Confidence Intervals)

The 95%-confidence interval is the interval $[-c; c]$ within which the true parameter value μ is situated with $\gamma = 95\%$ probability, i.e.

$$P\left(-c \leq \frac{\hat{\mu} - \mu}{\sigma/\sqrt{n}} \leq c\right) = \gamma \quad \text{or} \quad P\left(-c \leq \frac{\hat{\mu} - \mu}{\hat{\sigma}/\sqrt{n}} \leq c\right) = \gamma$$

(depending on whether we know the true variance of the distribution or not.)

To determine the critical value c, we use the *sample mean* $\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} X_i$ as an estimate of the true parameter value μ.

$$c = \Phi^{-1}\left(\frac{1+\gamma}{2}\right)$$

where Φ is the CDF of X.

a) *We know that $\sigma = 5.0$:*

Constructing a γ-*confidence interval* for the parameter μ

$$P(-c \leq \frac{\hat{\mu} - \mu}{\sigma/\sqrt{n}} \leq c) = \gamma$$

where $\frac{\hat{\mu}-\mu}{\sigma/\sqrt{n}} \sim \mathcal{N}(0,1)$.

The critical value

$$c = \Phi^{-1}\left(\frac{1-\gamma}{2}\right)$$

where Φ^{-1} is the inverse of the CDF of $\mathcal{N}(0,1)$. This is computed with the following Python code

```
25 gamma = 0.95
26 c = scipy.stats.norm.ppf((1+gamma)/2)
```

which gives us

$$c = 1.96$$

And the *sample mean*

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} X_i = 54.12$$

computed with the following Python code

```
17  x_mean = np.mean(x)
```

Finally, knowing the variance $\sigma = 5.0$ gives us the confidence interval for μ:

$$\left[\hat{\mu} - c\frac{\sigma}{\sqrt{n}}; \hat{\mu} + c\frac{\sigma}{\sqrt{n}}\right] = [38.63; 69.61]$$

with confidence level $\gamma = 95\%$.

b) *σ is unknown:*

In this case, we have

$$P\left(-c \leq \frac{\hat{\mu} - \mu}{S/\sqrt{n}} \leq c\right) = \gamma$$

where $S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \hat{\mu})^2$ is the *sample variance* and $\frac{\hat{\mu}-\mu}{S/\sqrt{n}} \sim t_{n-1}$, i.e. *t-distributed* with 1 degree of freedom.

The critical value is now computed using the inverse lookup of the CDF of the t-distribution with the following Python code

```
43  gamma = 0.95
44  c = scipy.stats.t.ppf((1+gamma)/2, n-1)
```

giving us

$$c = \Phi^{-1}\left(\frac{1+\gamma}{2}\right) = 2.26$$

Using the sample mean from before, we now get the following confidence for μ when the variance is unknown:

$$[36.24; 72.00]$$

with confidence level $\gamma = 95\%$.

The complete source code can be found in Appendix B. Below the output from running the Python script

```
Number of samples n:  10
a)
Known variance: 25.0
Sample mean: 54.12
Critical value c: 1.96
Confidence Interval: [ 38.63 ; 69.61 ]
b)
Sample variance S: 43.16
Critical value c: 2.26
Confidence Interval: [ 36.24 ; 72.00 ]
```

Regression

Question 4 (The Hot Air Balloon)

a) *The Mathematical Expression for the Data Matrix* $\mathbf{X}$*:*

Initially resorting to arbitrary, basic functions $h_0(x_n), \ldots, h_k(x_n)$, the idea is to "augment" all of the N data points with additional columns containing ... so as to get the $N \times K + 1$ dimensional data matrix

$$\mathbf{X} = \begin{bmatrix} h_0(x_1) & h_1(x_1) & \cdots & h_K(x_1) \\ h_0(x_2) & h_1(x_2) & \cdots & h_K(x_2) \\ \vdots & & & \\ h_0(x_N) & h_1(x_N) & \cdots & h_K(x_N) \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \\ \vdots \\ \mathbf{x}_N^\top \end{bmatrix}$$

to be used with the $K + 1$ dimensional parameter vector and the N dimensional target vector

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_K \end{bmatrix} \quad \text{and} \quad \mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_N \end{bmatrix}$$

The model can written as

$$f(x; \mathbf{w}) = \sum_{k=0}^{K} w_k h_k(x)$$

or using vector multiplication

$$f(\mathbf{x}_n, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}_n$$

The material is covered in R&G ch. 1.4 and in the L3 lecture. There appears to be some incongruency with regards to the dimensions and the numbering of the basic functions $h_k(x)$ in the book chapter and the lecture slides, where k sometimes start at 1 and sometimes start at 0. Starting from $k = 0$ is in line with the equation (1) provided in the exam question text.

K'th order polynomial: With $h_k(x) = x^k$ we get

$$f(x, \mathbf{w}) = \sum_{k=0}^{K} w_k h_k(x) = \sum_{k=0}^{K} w_k x^k$$

and the corresponding data matrix

$$\mathbf{X} = \begin{bmatrix} x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^K \\ x_2^0 & x_2^1 & x_2^2 & \cdots & x_2^K \\ \vdots & & & & \\ x_N^0 & x_N^1 & x_N^2 & \cdots & x_N^K \end{bmatrix}$$

Logarithmic model: In this case we have $h_0(x) = 1$ and $h_1(x) = \log x$, giving us the corresponding $N \times 2$ dimensional data matrix

$$\mathbf{X} = \begin{bmatrix} 1 & \log x_1 \\ 1 & \log x_2 \\ \vdots & \\ 1 & \log x_N \end{bmatrix}$$

and the 2-dimensional parameter vector

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$$

b) *Likelihood function for the data set:*

The general expression for our model is

$$t = f(x, \mathbf{w}) + \varepsilon = \sum_{k=0}^{K} w_k h_k(x) + \varepsilon$$

where $\varepsilon \sim \mathcal{N}(0, \sigma^2)$ is the additive noise.

The following material is especially covered in R&G Ch. 2.8.2.

Assuming that the experiments conducted are conditionally independent, we get the likehood function as the joint conditional density of all responses in the dataset

$$L = p(t_1, \ldots, t_N | x_1, \ldots, x_N, \mathbf{w}, \sigma^2)$$

expressed as

$$L = p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \sigma^2) = \prod_{n=1}^{N} p(t_n|x_n, \mathbf{w}, \sigma^2) = \prod_{n=1}^{N} \mathcal{N}(f(x_n, \mathbf{w}), \sigma^2)$$

where any dependency within the dataset is captured by the parameter $\mathbf{w}$ in the deterministic part of our model. Writing out the last expression we get the likelihood function

$$L = \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{ -\frac{1}{2}(t_n - f(x_n, \mathbf{w}))^2 \right\}$$

We can directly adopt the maximum likelihood solutions from R&G:

$$\widehat{\mathbf{w}} = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{t} \quad \text{and} \quad \widehat{\sigma^2} = \frac{1}{N}(\mathbf{t}^\top \mathbf{t} - \mathbf{t}^\top \mathbf{X}\widehat{\mathbf{w}})$$

The derivations can be found in ch. 2.8.2.

c) *Implementation in Python:*

Using the implemention of linear regression provided in the `linreg.py` from the L3 code, what is left do is simply augmenting the data points as stated above for the polynomial model and for the logarithmic model. However, I leave the prending of a columns of one's up to the `LinearRegression` class in `linreg.py`.

For the polynomial model simply use the `augment` function provided in `Non_Linear_Regression.ipynb` from the L3 Code without significant modifications

```
70  def augment(X, max_order):
71      """ Augments a given data
72      matrix by adding additional
73      columns.
74
75      NOTE: In case max_order is very large,
76      numerical inaccuracies might occur ...
77      """
78
79      X_augmented = X
80
81      for i in range(2, max_order+1):
82          X_augmented = np.concatenate([X_augmented, X**i], axis=1)
83
84      return X_augmented
```

For the logirithmic model simple using `np.log(X)` will suffice to transform the data matrix.

In both cases it is necessary to reshape the data matrix and the target vector

```
100 t = t.reshape((len(t),1))
101 X = X.reshape((len(X),1))
```

The estimated optimal parameter $\widehat{\mathbf{w}}$ can be computed using the `LinearRegression` class in `linreg.py` from L3 by fitting the linear model on the augmented/transformed datasets. For estimating the optimal parameter $\widehat{\sigma^2}$ additional computations must be done:

```
103 def optimal_sigma_squared(X,t,w_opt):
104     """
105     Maximum likelihood estimate for sigma_squared
106     """
107
108     # prepend a column of 1's
109     X = np.concatenate([np.ones(len(X)).reshape((len(X), 1)),X], axis=1)
110
111     #number of samples
112     N = len(X)
113
114     return 1/N * (t.T @ t - t.T @ X @ w_opt)
```

The complete script can be found in Appendix C. The `linreg.py` from L3 can be found in Appendix D.

d) *Applying the implementation from c) to the dataset:*

After reshaping the data matrix and the target vector, the implementation is put to work with the following code

```
116 # polynomial model
117 K = 3
118 X_polynomial = augment(X, K)
119 polynomial_model = linreg.LinearRegression()
120 polynomial_model.fit(X_polynomial, t)
121 polynomial_model.sigma_squared = optimal_sigma_squared(X_polynomial,
122                                                        t,
123                                                        polynomial_model.w)
124
125 # logarithmic model
126 X_logarithmic = np.log(X)
127 logarithmic_model = linreg.LinearRegression()
128 logarithmic_model.fit(X_logarithmic, t)
129 logarithmic_model.sigma_squared = optimal_sigma_squared(X_logarithmic,
130                                                         t,
131                                                         logarithmic_model.w)
```

For the polynomial model:

$$\widehat{\mathbf{w}} = \begin{bmatrix} 81.19 \\ 3.52 \\ -0.02 \\ 0.00 \end{bmatrix} \quad \text{and} \quad \widehat{\sigma^2} = 11.66$$

For the logarithmic model:

$$\widehat{\mathbf{w}} = \begin{bmatrix} 133.69 \\ 32.36 \end{bmatrix} \quad \text{and} \quad \widehat{\sigma^2} = 25.41$$

Figure 1 shows the estimated models plotted together with the data points of the dataset. I suspuct that the polynomial model might be overfitting data compared to the logarithmic model.

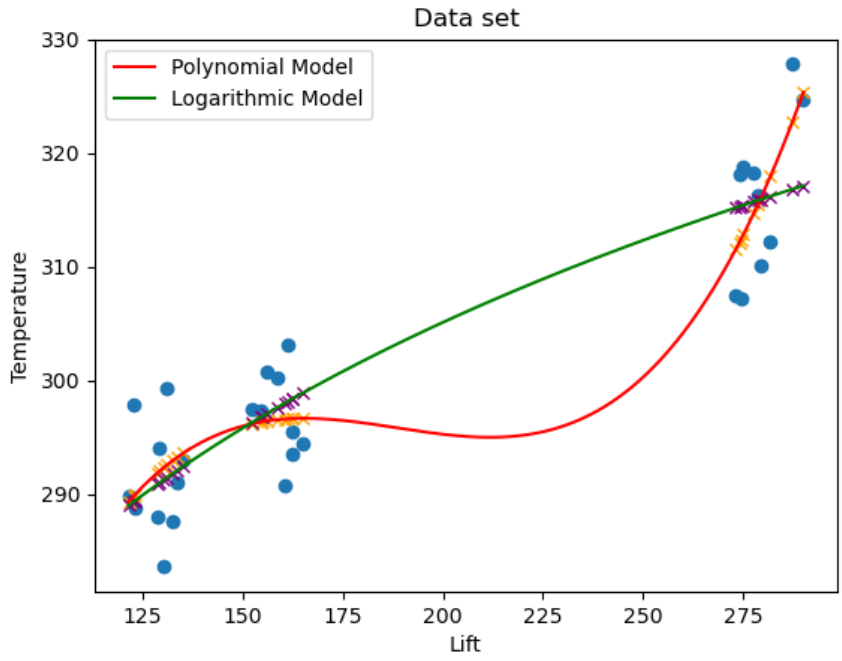


Figure 1: 4.d) Plotting the estimated models together with the data points.

e) *Implementing Bayesian Regression in Python*

Because of the conjugation of Guassian prior and likelihood the exact posterior density will also be Gaussian with

$$p(\mathbf{w}|\mathbf{t}, \mathbf{X}, \sigma^2) = \mathcal{N}(\boldsymbol{\mu}_w, \boldsymbol{\Sigma}_w)$$

where

$$\boldsymbol{\Sigma}_w = \left(\frac{1}{\sigma^2} \mathbf{X}^\top \mathbf{X} + \boldsymbol{\Sigma}_0^{-1} \right)^{-1}$$

and

$$\boldsymbol{\mu}_w = \boldsymbol{\Sigma}_w \left(\frac{1}{\sigma^2} \mathbf{X}^\top \mathbf{t} + \boldsymbol{\Sigma}_0^{-1} \boldsymbol{\mu}_0 \right)$$

The above is reused from my handin for A4. The same goes for the following functions which were also mainly developed as part of A4.

```
168 def Sigmaw(X, variance, Sigma_0):
169     '''
170     Compute the posterior Sigma_w
171     '''
172 
173     # prepend a column of 1's
174     X = np.concatenate([np.ones(len(X)).reshape((len(X), 1)),X], axis=1)
175 
176     return np.linalg.inv(1/variance * X.T @ X + np.linalg.inv(Sigma_0))
177 
178 def muw(X, t, variance, Sigma_0, mu_0, Sigma_w):
179     '''
180     Compute the posterior mu_w
181     '''
182 
183     # prepend a column of 1's
184     X = np.concatenate([np.ones(len(X)).reshape((len(X), 1)),X], axis=1)
185 
186     return Sigma_w @ (1/variance * X.T @ t + np.linalg.inv(Sigma_0) @ mu_0)
```

f) Putting the implementation into action for the polynomial model

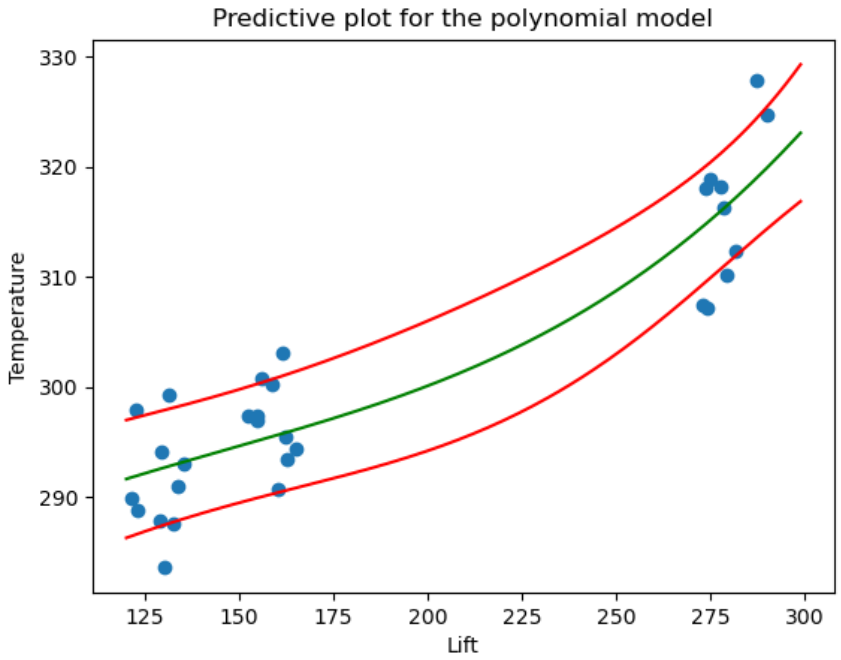
```
193 mu_0_polynomial = np.array([268,0,0,0]).reshape((4,1))
194 Sigma_0_polynomial = np.eye(K+1) * sigma_squared_0
195 Sigma_w_polynomial = Sigmaw(X_polynomial, variance, Sigma_0_polynomial)
196 mu_w_polynomial = muw(X_polynomial, t, variance, Sigma_0_polynomial, mu_0_polynomial,
197                       Sigma_w_polynomial)
198 
199 xnew = xnew.reshape((len(xnew),1))
200 xnew = augment(xnew, K)
201 xnew = np.concatenate([np.ones(len(xnew)).reshape((len(xnew), 1)), xnew], axis=1)
202 
203 mu_pred = xnew @ mu_w_polynomial
204 
205 sigma2_pred = np.diag(variance + xnew @ Sigma_w_polynomial @ xnew.T)
206 sigma2_pred = sigma2_pred.reshape((len(sigma2_pred), 1))
```

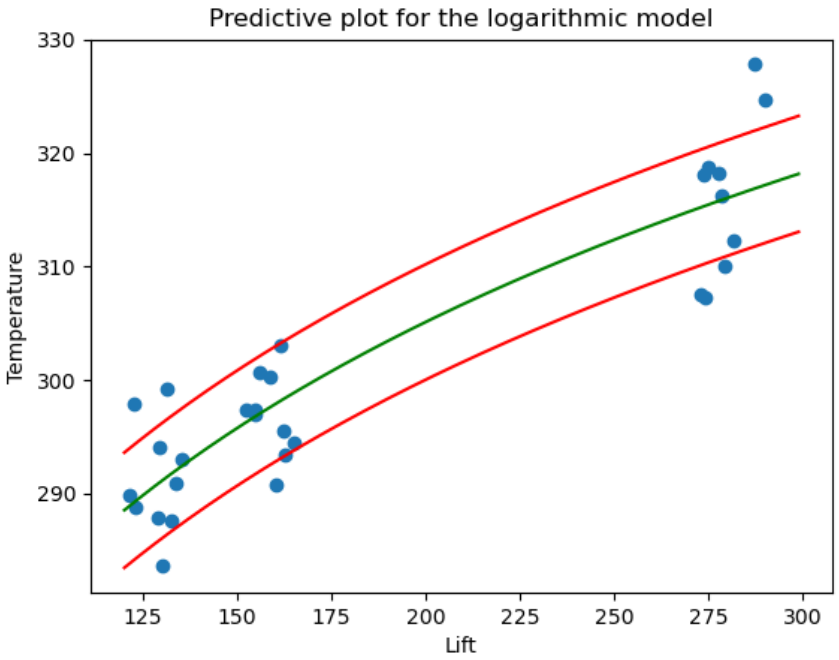
and for the logarithmic model

```
214 mu_0_logarithmic = np.array([133,32]).reshape((2,1))
215 Sigma_0_logarithmic = np.eye(2) * sigma_squared_0
216 Sigma_w_logarithmic = Sigmaw(X_logarithmic, variance, Sigma_0_logarithmic)
217 mu_w_logarithmic = muw(X_logarithmic, t, variance, Sigma_0_logarithmic,
218                        mu_0_logarithmic,
```

```
                        Sigma_w_logarithmic)

xnew = xnew[:,1].reshape((len(xnew[:,1]),1))
xnew_log = np.log(xnew)
xnew_log = np.concatenate([np.ones(len(xnew_log)).reshape(
    (len(xnew_log), 1)), xnew_log], axis=1)

mu_pred = xnew_log @ mu_w_logarithmic

sigma2_pred = np.diag(variance + xnew_log @ Sigma_w_logarithmic @ xnew_log.T)
sigma2_pred = sigma2_pred.reshape((len(sigma2_pred), 1))
```

Below is shown the plotted predictive distributions. Using Bayesian regression, we see much less over-fitting for the polynomial model.




Predictive plot for the logarithmic model
Temperature
Lift
330
320
310
300
290
125
150
175
200
225
250
275
300

Classification

Question 5 (Classification & Random Forests)

Labeling the original set T and the two subsets T_0 and T_1, respectively. The total number of samples is $12+5+8+9=34$ in T and there are 19 and 15 sample in T_0 and T_1, respectively.

a) *Entropy:*

$$H(T) = -\frac{12}{34}\log_2\left(\frac{12}{34}\right) - \frac{5}{34}\log_2\left(\frac{5}{34}\right) - \frac{8}{34}\log_2\left(\frac{8}{34}\right) - \frac{9}{34}\log_2\left(\frac{9}{34}\right) = 1.936$$

$$H(T_0) = -\frac{6}{19}\log_2\left(\frac{6}{19}\right) - \frac{1}{19}\log_2\left(\frac{1}{19}\right) - \frac{6}{19}\log_2\left(\frac{6}{19}\right) - \frac{6}{19}\log_2\left(\frac{6}{19}\right) = 1.799$$

and similarly we get

$$H(T_1) = -\sum_{c\in C} p(c)\log_2 p(c) = 1.889$$

where $p(c) = \frac{\#\text{samples in class}}{\#\text{total samples}}$.

The information gain is computed

$$\begin{aligned} Gain(T,j) &= H(T) - \sum_{i=\{0,1\}} \frac{|T_i|}{|T|} H(T_i) \\ &= 1.936 - \frac{19}{34}\times 1.799 - \frac{15}{34}\times 1.889 \\ &= 0.097 \end{aligned}$$

where j is just refering to the particular split.

b) *Gini:*

$$Gini(T,j) = gini(T) - \frac{19}{34}\times gini(T_0) - \frac{15}{34}\times gini(T_1)$$

where j is just refering to the particular split and

$$gini(T) = 1 - \sum_i \left(\frac{|T_i|}{|T|}\right)^2$$

Thus, we have

$$\begin{aligned} Gini(T,j) &= 0.73 - \frac{19}{34}\times 0.70 - \frac{15}{34}\times 0.71 \\ &= 0.025 \end{aligned}$$

Question 6 (Classification & Validation)

The complete source code for this question can be found in Appendix E.

a) *Implementing random forest training using* `sklearn.ensemble.RandomForestClassifier`
Loading the relevant data:

```
13 data_train = np.loadtxt(
14     '../data/accent-mfcc-data_shuffled_train.txt',
15     delimiter=',')
16 data_validation = np.loadtxt(
17     '../data/accent-mfcc-data_shuffled_validation.txt',
18     delimiter=',')
19
20 t_train = data_train[:,0]
21 X_train = data_train[:,1:]
22
23 t_validation = data_validation[:,0]
24 X_validation = data_validation[:,1:]
```

Setting up the classifier, training it using the training data, and using it to predict—see Listing 1.

b) I am using `ParameterGrid` from `sklearn.model_selection` to generate a list of all possible permutations of the specified parameters used for iteration to perform the optimization search. For each iteration the parameters and the resulting performance metrics are added to a list, which is easily sorted. Bulat suggested using the `accuracy_score` function on Absalon instead of the actual number of correctly classified samples as the first metric, so this is implemented.

See Listing 1.

c) *Results*

The performance increases as we increase tree depth, the number of features considered for each split, and the complexity of the criterion used (*gini* or *entropy*)—but the computations also become more complex. Listing 2 show the report from the optimization loop. For a more conventient overview I also provide a sorted print of the permutated parameters as they yield better and better metrics, see Appendix F.

We get optimal results with parameters

```
{'criterion': 'entropy', 'max_depth': 10, 'max_features': 'sqrt'}
```

```
param_grid = {
    'criterion'    : ['gini', 'entropy'],
    'max_depth'    : [2,5,7,10,15],
    'max_features' : ['sqrt', 'log2']
    }

res = np.empty((0,3)) # for resulting metrics

for params in list(ParameterGrid(param_grid)):
    # setup classifier using parameters
    clf = RandomForestClassifier(
        criterion    = params['criterion'],
        max_depth    = params['max_depth'],
        max_features = params['max_features'])

    # train
    clf.fit(X_train, t_train)

    # number of correctly classified validation samples
    t_pred = clf.predict(X_validation)
    acc_score = accuracy_score(t_validation, t_pred)

    # probability associated with classification
    t_prob = clf.predict_proba(X_validation)
    prob_score = np.mean([t_prob[int(t_val)]
                          for (t_prob, t_val)
                          in zip(t_prob, t_validation)])

    print("Accuracy score: %.2f"
          %acc_score)
    print("Average probability assigned to correct classes: %.2f"
        %prob_score)

    # print params if more optimal than previously tried
    if len(res) > 0 and (acc_score > res[-1,1]
                         or (acc_score == res[-1,1]
                             and prob_score > res[-1,2])):
        print(params)

    # accumulate results
    res = np.append(res, np.array([[params, acc_score, prob_score]]), axis=0)
    res = res[np.lexsort((res[:,1], res[:,2]))] # sort ascending
```

Listing 1: a) and b): random forest implementation and optimazation loop.

```
Accuracy score: 0.48
Average probability assigned to correct classes: 0.36
Accuracy score: 0.52
Average probability assigned to correct classes: 0.37
{'criterion': 'gini', 'max_depth': 2, 'max_features': 'log2'}
Accuracy score: 0.68
Average probability assigned to correct classes: 0.48
{'criterion': 'gini', 'max_depth': 5, 'max_features': 'sqrt'}
Accuracy score: 0.68
Average probability assigned to correct classes: 0.49
{'criterion': 'gini', 'max_depth': 5, 'max_features': 'log2'}
Accuracy score: 0.75
Average probability assigned to correct classes: 0.53
{'criterion': 'gini', 'max_depth': 7, 'max_features': 'sqrt'}
Accuracy score: 0.74
Average probability assigned to correct classes: 0.52
Accuracy score: 0.79
Average probability assigned to correct classes: 0.57
{'criterion': 'gini', 'max_depth': 10, 'max_features': 'sqrt'}
Accuracy score: 0.78
Average probability assigned to correct classes: 0.56
Accuracy score: 0.78
Average probability assigned to correct classes: 0.57
Accuracy score: 0.78
Average probability assigned to correct classes: 0.57
Accuracy score: 0.51
Average probability assigned to correct classes: 0.38
Accuracy score: 0.53
Average probability assigned to correct classes: 0.37
Accuracy score: 0.77
Average probability assigned to correct classes: 0.52
Accuracy score: 0.74
Average probability assigned to correct classes: 0.51
Accuracy score: 0.74
Average probability assigned to correct classes: 0.56
Accuracy score: 0.79
Average probability assigned to correct classes: 0.55
{'criterion': 'entropy', 'max_depth': 7, 'max_features': 'log2'}
Accuracy score: 0.81
Average probability assigned to correct classes: 0.59
{'criterion': 'entropy', 'max_depth': 10, 'max_features': 'sqrt'}
Accuracy score: 0.82
Average probability assigned to correct classes: 0.57
{'criterion': 'entropy', 'max_depth': 10, 'max_features': 'log2'}
Accuracy score: 0.81
Average probability assigned to correct classes: 0.57
Accuracy score: 0.82
Average probability assigned to correct classes: 0.58
```


Listing 2: c) Report from optimization loop.

Clustering

Question 7 (K-means Clustering & Principal Component Analisys)

The complete source code can be found in Appendix **??**.

a) *Normalizing the data*

```
9
10  def normalize(M):
11      """
12      Normalize all columns in 2D Numpy-array M
13
14      Returns the normalized M
15      """
```

b) *Implementing K-means clustering*

The `k_mean_clstr` function shown implements the algorithm. Utility functions—as well as the rest of the source code—can be found in Appendix G.

```
143
144 def k_mean_clstr(k, data, centroids):
145     """
146     K-means clustering
147
148     Params:
149     ------
150     k : number of clusters
151     data : (n_samples, n_features) (normalized) data matrix
152     rng  : random generator
153
154     Returns:
155     --------
156     assignments, intra_cluster_dist : tuple
157
158     """
159     assignments = assign_datapoints_to_centroids(data, centroids)
160
161     new_assignments = [] # initial dummy val
162
163     while not np.array_equal(assignments, new_assignments):
164         # repeat until assignments does not change
165         centroids = compute_new_centroids(data, assignments, centroids)
166         assignments = new_assignments
167         new_assignments = assign_datapoints_to_centroids(data, centroids)
168
```

```
169     intra_cluster_dist = compute_sum_intra_cluster_dist(data, assignments, centroids)
170
```

c) *Solution* With $K = 3$ and five runs of the algorithm yields the following results:

```
Smallest Intra-Cluster Distance: 287.2202
Number of samples in cluster 0: 71
Number of samples in cluster 1: 70
Number of samples in cluster 2: 69
```

The implementation uses a fixed seed for the random generator, so the results can be reproduced simply running the `q7_clustering.py`

d) *PCA* Reusing the relevant functions implemented (or alreadily provided) as part of A5. Only the `__visualizeLabels()` function has been modified to accomodate for the centroids also. The code snippet below snow the dimension reduction of the clustered data and centroids from question c).

```
31  PCevals, PCevecs = __PCA(features)
32
33  # Convert data to two dimemsions using PCA
34  features2D = __transformData(features, PCevecs)
35  centroids2D = __transformData(centroids, PCevecs)
```

The following line has been added to the `__visualizeLabels()` function:

```
48      plt.scatter(centroids[:, 0], centroids[:,1], c = 'black', s=100)
```

The complete source code can be found in Appendix H.

e) *Plot* See snippet above.

Figure 2 show the plotted clustering after PCA. The plot display a nice seperation samples into clusters centered around the centroids. However, two neighboring samples that have been assigned to different clusters, i.e. one colored red and the other blue. Using a more sophisticated distance metric than the Euclidean distance might address this.

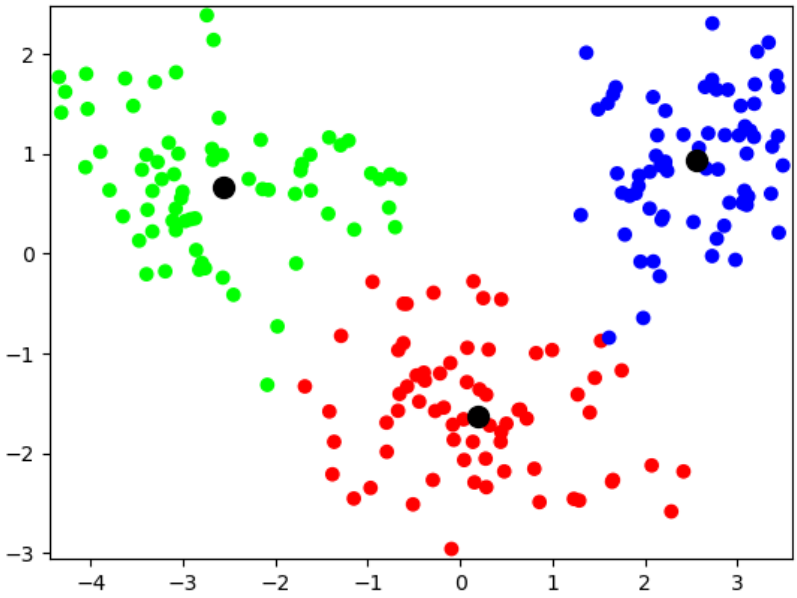


Figure 2: Plot of clustering after PCA

A q2_hypothesis-testing.py

```python
#!/usr/bin/env python
#
# MAD 2020-21, Exam
# Question 2 (Hypothesis Testing)
#

import numpy as np
import scipy.stats

# known variance
sigma_squared = 1.0

# samples
X = np.array([8.2, 7.9, 8.7, 8.3, 8.5, 8.3, 8.8, 8.2, 8.7, 7.6, 8.4])
n = len(X)

# sample mean
X_mean = np.mean(X)

# z-test
mu = 8.5
alpha = 0.05
z = np.sqrt(n) * (X_mean - mu) / np.sqrt(sigma_squared)
c = scipy.stats.norm.ppf(alpha)

print("Known variance: %.1f" %sigma_squared)
print("Samples:", X)
print("Number of samples:", n)
print("Sample mean x: %.4f" %X_mean)
print("Significance Level alpha: %.2f" %alpha)
print("Test statistics for the samples z: %.4f" %z)
print("Critical Value: %.4f" %c)
```

B q3_confidence-intervals.py

```python
#!/usr/bin/env python
#
# MAD 2020-21, EXAM
# Question 3 (Confidence Intervals)
#

import numpy as np
import scipy.stats

# samples from normal distributed X \sim N(mu, sigma_squared)
x = np.array([56.6, 59.0, 53.2, 66.1, 51.3, 50.4, 53.5, 44.5, 46.3, 60.3])
n = len(x) # number of samples

print("Number of samples n: ", n)

# sample mean
x_mean = np.mean(x)

#
# a)
#
# known variance
sigma_squared = 5.0**2

gamma = 0.95
c = scipy.stats.norm.ppf((1+gamma)/2)

ac = x_mean - c * sigma_squared / np.sqrt(n)
bc = x_mean + c * sigma_squared / np.sqrt(n)

print("a)")
print("Known variance: %.1f" %sigma_squared)
print("Sample mean: %.2f" %x_mean)
print("Critical value c: %.2f" %c)
print("Confidence Interval: [ %.2f ; %.2f ]" %(ac, bc))

#
# b)
#
# sample variance
S = np.mean(np.var(x, ddof=1))

gamma = 0.95
c = scipy.stats.t.ppf((1+gamma)/2, n-1)

ac = x_mean - c * sigma_squared / np.sqrt(n)
bc = x_mean + c * sigma_squared / np.sqrt(n)

print("b)")
```

```
50  print("Sample variance S: %.2f" %S)
51  print("Critical value c: %.2f" %c)
52  print("Confidence Interval: [ %.2f ; %.2f ]" %(ac, bc))
```

C regression.py

```python
#!/usr/bin/env python

import numpy as np
import matplotlib.pyplot as plt


def loaddata(filename):
    """Load the balloon data set from filename and return t, X
        t - N-dim. vector of target (temperature) values
        X - N-dim. vector containing the inputs (lift) x for each data point
    """
    # Load data set from CSV file
    Xt = np.loadtxt(filename, delimiter=',')

    # Split into data matrix and target vector
    X = Xt[:,0]
    t = Xt[:,1]

    return t, X


def predictiveplot(xnew, mu_pred, sigma2_pred, t, X):
    """Plots the mean of the predictive distribution (green curve) and +/- the predictive standard d
        xnew - Mx1 vector of new input x values to evaluate the predictive distribution for
        mu_pred - Mx1 vector of predictive mean values evaluated at xnew,
        sigma2_pred - Mx1 vector of predictive standard deviation values evaluated at xnew
        t - vector containing the target values of the training data set
        X - vector containing the input values of the training data set
    """
    plt.figure()
    plt.scatter(X, t)
    plt.plot(xnew, mu_pred, 'g')
    plt.plot(xnew, mu_pred + np.sqrt(sigma2_pred).reshape((sigma2_pred.shape[0],1)), 'r')
    plt.plot(xnew, mu_pred - np.sqrt(sigma2_pred).reshape((sigma2_pred.shape[0],1)), 'r')



# Load data
t, X = loaddata('../data/hot-balloon-data.csv')


# Visualize the data
plt.figure()
plt.scatter(X, t)
plt.xlabel('Lift')
plt.ylabel('Temperature')
plt.title('Data set')


```

```
50
51  # This is a good range of input x values to use when visualizing the estimated models
52  xnew = np.arange(120, 300, dtype=np.float)
53
54  # # Example of how to use the predictiveplot function
55  # mu_fake = 0.25 * xnew.reshape((xnew.shape[0],1)) + 250.0
56  # sigma2_fake = mu_fake
57  # predictiveplot(xnew, mu_fake, sigma2_fake, t, X)
58  # plt.xlabel('Lift')
59  # plt.ylabel('Temperature')
60  # plt.title('Example of predictiveplot')
61
62
63
64  # ADD YOUR SOLUTION CODE HERE!
65
66  # import linear regression implementation provided in L3
67  import linreg
68
69  #
70  def augment(X, max_order):
71      """ Augments a given data
72      matrix by adding additional
73      columns.
74
75      NOTE: In case max_order is very large,
76      numerical inaccuracies might occur ...
77      """
78
79      X_augmented = X
80
81      for i in range(2, max_order+1):
82          X_augmented = np.concatenate([X_augmented, X**i], axis=1)
83
84      return X_augmented
85
86
87  def logarithmic(X):
88      """Augments a given N x 1 data matrix by prepending
89      a column of 1's and wrapping each x_n in a log-function.
90
91      Returns the augmented N x 2 data matrix for the logarithmic model
92      """
93      # return np.concatenate([np.ones(len(X)).reshape((len(X), 1)),
94      #                        np.log(X)], axis=1)
95      return np.log(X)
96
97
98  # reshape both arrays to make sure that we deal with
99  # N-dimensional Numpy arrays
100 t = t.reshape((len(t),1))
```

```
X = X.reshape((len(X),1))

def optimal_sigma_squared(X,t,w_opt):
    """
    Maximum likelihood estimate for sigma_squared
    """

    # prepend a column of 1's
    X = np.concatenate([np.ones(len(X)).reshape((len(X), 1)),X], axis=1)

    #number of samples
    N = len(X)

    return 1/N * (t.T @ t - t.T @ X @ w_opt)

# polynomial model
K = 3
X_polynomial = augment(X, K)
polynomial_model = linreg.LinearRegression()
polynomial_model.fit(X_polynomial, t)
polynomial_model.sigma_squared = optimal_sigma_squared(X_polynomial,
                                                       t,
                                                       polynomial_model.w)

# logarithmic model
X_logarithmic = np.log(X)
logarithmic_model = linreg.LinearRegression()
logarithmic_model.fit(X_logarithmic, t)
logarithmic_model.sigma_squared = optimal_sigma_squared(X_logarithmic,
                                                        t,
                                                        logarithmic_model.w)

# printing
print("Polynomial model:")
print("Optimal w: \n %s" %str(polynomial_model.w))
print("Optimal sigma_squared:\n %s \n" %str(polynomial_model.sigma_squared))

print("Logarithmic model:")
print("Optimal w:\n %s" %str(logarithmic_model.w))
print("Optimal sigma_squared:\n %s\n" %str(logarithmic_model.sigma_squared))

# figures
# Code from Non_Linear_Regression.ipynb
Xplot = np.arange(X.min(), X.max(), 0.01)
Xplot = Xplot.reshape((len(Xplot),1))
Xplot = augment(Xplot, K)
pred_plot = polynomial_model.predict(Xplot)

polynomial_pred = polynomial_model.predict(X_polynomial)
logarithmic_pred = logarithmic_model.predict(X_logarithmic)

```

```
plt.plot(X, polynomial_pred, 'x', color='orange')
plt.plot(X, logarithmic_pred, 'x', color='purple')
plt.plot(Xplot[:,0], pred_plot, '-', color='red', label="Polynomial Model")

Xplot = np.arange(X.min(), X.max(), 0.01)
Xplot = Xplot.reshape((len(Xplot),1))
pred_log_plot = logarithmic_model.predict(np.log(Xplot))

plt.plot(Xplot[:,0], pred_log_plot, '-', color='green', label="Logarithmic Model")
plt.legend()

#
# Bayesian regression
#
# Code developed in A4

def Sigmaw(X, variance, Sigma_0):
    '''
    Compute the posterior Sigma_w
    '''

    # prepend a column of 1's
    X = np.concatenate([np.ones(len(X)).reshape((len(X), 1)),X], axis=1)

    return np.linalg.inv(1/variance * X.T @ X + np.linalg.inv(Sigma_0))

def muw(X, t, variance, Sigma_0, mu_0, Sigma_w):
    '''
    Compute the posterior mu_w
    '''

    # prepend a column of 1's
    X = np.concatenate([np.ones(len(X)).reshape((len(X), 1)),X], axis=1)

    return Sigma_w @ (1/variance * X.T @ t + np.linalg.inv(Sigma_0) @ mu_0)

# parameters used for both models
variance = 25 # likelihood variance
sigma_squared_0 = 10

# polynomial_model
mu_0_polynomial = np.array([268,0,0,0]).reshape((4,1))
Sigma_0_polynomial = np.eye(K+1) * sigma_squared_0
Sigma_w_polynomial = Sigmaw(X_polynomial, variance, Sigma_0_polynomial)
mu_w_polynomial = muw(X_polynomial, t, variance, Sigma_0_polynomial, mu_0_polynomial,
                      Sigma_w_polynomial)

xnew = xnew.reshape((len(xnew),1))
xnew = augment(xnew, K)
xnew = np.concatenate([np.ones(len(xnew)).reshape((len(xnew), 1)), xnew], axis=1)

```

```
mu_pred = xnew @ mu_w_polynomial

sigma2_pred = np.diag(variance + xnew @ Sigma_w_polynomial @ xnew.T)
sigma2_pred = sigma2_pred.reshape((len(sigma2_pred), 1))

predictiveplot(xnew[:,1], mu_pred, sigma2_pred, t, X)
plt.xlabel('Lift')
plt.ylabel('Temperature')
plt.title('Predictive plot for the polynomial model')

# logarithmic model
mu_0_logarithmic = np.array([133,32]).reshape((2,1))
Sigma_0_logarithmic = np.eye(2) * sigma_squared_0
Sigma_w_logarithmic = Sigmaw(X_logarithmic, variance, Sigma_0_logarithmic)
mu_w_logarithmic = muw(X_logarithmic, t, variance, Sigma_0_logarithmic,
                       mu_0_logarithmic,
                       Sigma_w_logarithmic)

xnew = xnew[:,1].reshape((len(xnew[:,1]),1))
xnew_log = np.log(xnew)
xnew_log = np.concatenate([np.ones(len(xnew_log)).reshape(
    (len(xnew_log), 1)), xnew_log], axis=1)

mu_pred = xnew_log @ mu_w_logarithmic

sigma2_pred = np.diag(variance + xnew_log @ Sigma_w_logarithmic @ xnew_log.T)
sigma2_pred = sigma2_pred.reshape((len(sigma2_pred), 1))

predictiveplot(xnew, mu_pred, sigma2_pred, t, X)
plt.xlabel('Lift')
plt.ylabel('Temperature')
plt.title('Predictive plot for the logarithmic model')


# Show all figures
plt.show()
```

D linreg.py

```python
import numpy

# NOTE: This template makes use of Python classes. If
# you are not yet familiar with this concept, you can
# find a short introduction here:
# http://introtopython.org/classes.html

class LinearRegression():
    """
    Linear regression implementation.
    """

    def __init__(self, lam=0.0):

        self.lam = lam

    def fit(self, X, t):
        """
        Fits the linear regression model.

        Parameters
        ----------
        X : Array of shape [n_samples, n_features]
        t : Array of shape [n_samples, 1]
        """

        # make sure that we have Numpy arrays; also
        # reshape the target array to ensure that we have
        # a N-dimensional Numpy array (ndarray), see
        # https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.ndarray.html
        X = numpy.array(X).reshape((len(X), -1))
        t = numpy.array(t).reshape((len(t), 1))

        # prepend a column of ones
        ones = numpy.ones((X.shape[0], 1))
        X = numpy.concatenate((ones, X), axis=1)

        # compute weights (solve system)
        diag = self.lam * len(X) * numpy.identity(X.shape[1])
        a = numpy.dot(X.T, X) + diag
        b = numpy.dot(X.T, t)
        self.w = numpy.linalg.solve(a,b)

    def predict(self, X):
        """
        Computes predictions for a new set of points.

        Parameters
        ----------
```

```
50          X : Array of shape [n_samples, n_features]
51
52          Returns
53          -------
54          predictions : Array of shape [n_samples, 1]
55          """
56
57          # make sure that we have Numpy arrays; also
58          # reshape the target array to ensure that we have
59          # a N-dimensional Numpy array (ndarray), see
60          # https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.ndarray.html
61          X = numpy.array(X).reshape((len(X), -1))
62
63          # prepend a column of ones
64          ones = numpy.ones((X.shape[0], 1))
65          X = numpy.concatenate((ones, X), axis=1)
66
67          # compute predictions
68          predictions = numpy.dot(X, self.w)
69
70          return predictions
```

E q6_random-forests.py

```python
#!/usr/bin/env python
#
# MAD 2020-21, Exam
# Question 6, Classification & Random Forests
#

import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import ParameterGrid
from sklearn.metrics import accuracy_score

# load data
data_train = np.loadtxt(
    '../data/accent-mfcc-data_shuffled_train.txt',
    delimiter=',')
data_validation = np.loadtxt(
    '../data/accent-mfcc-data_shuffled_validation.txt',
    delimiter=',')

t_train = data_train[:,0]
X_train = data_train[:,1:]

t_validation = data_validation[:,0]
X_validation = data_validation[:,1:]

print("Shape of training data: %s" %str(data_train.shape))
print("Shape of validation data: %s"%str(data_validation.shape))
print("Shape of training targets: %s" %str(t_train.shape))
print("Shape of training features: %s" %str(X_train.shape))
print("Shape of validation targets: %s" %str(t_validation.shape))
print("Shape of validation features: %s" %str(X_validation.shape))
print()

# b) finding optimal set of random forest classifier parameters
param_grid = {
    'criterion'    : ['gini', 'entropy'],
    'max_depth'    : [2,5,7,10,15],
    'max_features' : ['sqrt', 'log2']
    }

res = np.empty((0,3)) # for resulting metrics

for params in list(ParameterGrid(param_grid)):
    # setup classifier using parameters
    clf = RandomForestClassifier(
        criterion    = params['criterion'],
        max_depth    = params['max_depth'],
        max_features = params['max_features'])

```

```
50      # train
51      clf.fit(X_train, t_train)
52
53      # number of correctly classified validation samples
54      t_pred = clf.predict(X_validation)
55      acc_score = accuracy_score(t_validation, t_pred)
56
57      # probability associated with classification
58      t_prob = clf.predict_proba(X_validation)
59      prob_score = np.mean([t_prob[int(t_val)]
60                            for (t_prob, t_val)
61                            in zip(t_prob, t_validation)])
62
63      print("Accuracy score: %.2f"
64            %acc_score)
65      print("Average probability assigned to correct classes: %.2f"
66          %prob_score)
67
68      # print params if more optimal than previously tried
69      if len(res) > 0 and (acc_score > res[-1,1]
70                           or (acc_score == res[-1,1]
71                               and prob_score > res[-1,2])):
72          print(params)
73
74      # accumulate results
75      res = np.append(res, np.array([[params, acc_score, prob_score]]), axis=0)
76      res = res[np.lexsort((res[:,1], res[:,2]))] # sort ascending
77
78  for t in res:
79      print(t[0])
```

F Q6.c: Sorted parameters

A sorted print of the permutated parameters as the yield better and better metrics.

```
{'criterion': 'entropy', 'max_depth': 15, 'max_features': 'log2'}
{'criterion': 'gini', 'max_depth': 2, 'max_features': 'sqrt'}
{'criterion': 'gini', 'max_depth': 2, 'max_features': 'log2'}
{'criterion': 'entropy', 'max_depth': 2, 'max_features': 'log2'}
{'criterion': 'entropy', 'max_depth': 2, 'max_features': 'sqrt'}
{'criterion': 'gini', 'max_depth': 5, 'max_features': 'sqrt'}
{'criterion': 'gini', 'max_depth': 5, 'max_features': 'log2'}
{'criterion': 'entropy', 'max_depth': 5, 'max_features': 'log2'}
{'criterion': 'entropy', 'max_depth': 5, 'max_features': 'sqrt'}
{'criterion': 'gini', 'max_depth': 7, 'max_features': 'log2'}
{'criterion': 'gini', 'max_depth': 7, 'max_features': 'sqrt'}
{'criterion': 'entropy', 'max_depth': 7, 'max_features': 'log2'}
{'criterion': 'entropy', 'max_depth': 7, 'max_features': 'sqrt'}
{'criterion': 'gini', 'max_depth': 10, 'max_features': 'log2'}
{'criterion': 'gini', 'max_depth': 10, 'max_features': 'sqrt'}
{'criterion': 'gini', 'max_depth': 15, 'max_features': 'log2'}
{'criterion': 'gini', 'max_depth': 15, 'max_features': 'sqrt'}
{'criterion': 'entropy', 'max_depth': 15, 'max_features': 'sqrt'}
{'criterion': 'entropy', 'max_depth': 10, 'max_features': 'log2'}
{'criterion': 'entropy', 'max_depth': 15, 'max_features': 'log2'}
{'criterion': 'entropy', 'max_depth': 10, 'max_features': 'sqrt'}
```

G q7_clustering.py

```python
#!/usr/bin/env python
#
# MAD 2020-21, Exam number: 23
# Question 7 (Clustering)
#

import numpy as np


def normalize(M):
    """
    Normalize all columns in 2D Numpy-array M

    Returns the normalized M
    """
    return (M - np.mean(M, axis=0)) / np.std(M, axis=0)


def generate_seed_centroids(k, data, rng):
    """
    Randomly choose k datapoints from data matrix to be used as seed centroids

    Parameters:
    -----------
    k   : int number of centroids
    data   : (n_samples, n_features) datamatrix
    rng : np random generator

    Returns:
    --------
    centroids : Numpy array of k random centroids
    """
    centroids = np.empty((0,data.shape[1]))

    for _ in range(k):
        i = int(rng.random() * data.shape[0])
        datapoint = data[i]
        centroids = np.vstack((centroids, datapoint))

    return centroids

def dist(a,b):
    """
    Compute Euclidean distance between two points
    """
    return np.linalg.norm(a-b)

def get_nearest_centroid(datapoint, centroids):
    """
```

```
    Computes the index of the nearest centroid

    Params:
    ------
    datapoint : (n_features) np.array
    centroid  : (k, n_features) np.array of centroids

    Returns:
    -------
    index of neares centroid

    """
    distances = [dist(datapoint, centroid) for centroid in centroids]
    return np.argsort(distances)[0]

def assign_datapoints_to_centroids(data, centroids):
    """
    Assign datapoints to nearest centroids

    Params:
    -------
    data      : (n_samples, n_features) data matrix
    centroids : (k, n_features) array of centroids

    Returns:
    --------
    assignments : np.array of indices mapping each datapoint to
    its nearest centroid
    """
    assignments = [get_nearest_centroid(datapoint, centroids) for datapoint in data]
    return np.array(assignments)

def compute_sum_intra_cluster_dist(data, assignments, centroids):
    """
    Compute the sum of intra cluster distances of all k clusters

    Params:
    -------
    data          : (n_samples, n_features) data matrix
    assignments   : (n_samples) np.array
    centroids     : (k, n_features)


    Returns:
    -------
    computed intra cluster distance
    """
    k = len(centroids)

    # 3D-array of k clusters with assigned datapoints
    clusters = np.array([data[np.where(assignments == j)] for j in range(k)])
```

```
101
102     sum = 0
103
104     for j in range(k):
105         sum += compute_intra_cluster_dist(clusters[j], centroids[j])
106
107     return sum
108
109 def compute_intra_cluster_dist(cluster, centroid):
110     """
111     Compute the intra cluster distance of a single cluster
112     """
113     return np.sum([dist(datapoint, centroid) for datapoint in cluster])
114
115 def compute_new_centroids(data, assignments, centroids):
116     """
117     Compute new centroids
118
119     Params:
120     -------
121     data          : (n_samples, n_features) data matrix
122     assignments   : (n_samples) np.array
123     centroids     : (k, n_features)
124
125     Returns:
126     --------
127     centroids : (k, n_features) np.array
128        The new centroids
129     """
130     k = len(centroids)
131
132     # 3D-array of k clusters with assigned datapoints
133     clusters = np.array([data[np.where(assignments == j)] for j in range(k)])
134
135     for j in range(k):
136         # number of datapoints in j'th cluster
137         n = clusters[j].shape[0]
138         if (n > 0):
139             # update j'th centroid
140             centroids[j] = 1/n * np.sum(clusters[j], axis=0)
141
142     return centroids
143
144 def k_mean_clstr(k, data, centroids):
145     """
146     K-means clustering
147
148     Params:
149     ------
150     k : number of clusters
151     data : (n_samples, n_features) (normalized) data matrix
```

```
    rng  : random generator

    Returns:
    --------
    assignments, intra_cluster_dist : tuple

    """
    assignments = assign_datapoints_to_centroids(data, centroids)

    new_assignments = [] # initial dummy val

    while not np.array_equal(assignments, new_assignments):
        # repeat until assignments does not change
        centroids = compute_new_centroids(data, assignments, centroids)
        assignments = new_assignments
        new_assignments = assign_datapoints_to_centroids(data, centroids)

    intra_cluster_dist = compute_sum_intra_cluster_dist(data, assignments, centroids)

    return assignments, intra_cluster_dist


# load data
data = np.loadtxt('../data/seedsDataset.txt', delimiter=',')

print("Shape of data: %s" %str(data.shape))

# K-means clustering


# test
data_norm = normalize(data)

seed = 111
rng = np.random.default_rng(seed)

k = 3 # numbers of clusters

# running K-means clustering 5 times
# solution with smallest intra-cluster distance

res = np.empty((0,3))

for i in range(5):
    centroids = generate_seed_centroids(k,data_norm, rng)
    assignments, intra_cluster_dist = k_mean_clstr(k, data_norm, centroids)

    res = np.vstack((res, [centroids, assignments, intra_cluster_dist]))

# get results

```

```
203 centroids, assignments, intra_cluster_dist = res[np.argsort(res[:,2])[0]]
204
205 print("Smallest Intra-Cluster Distance: %.4f" %intra_cluster_dist)
206
207 # 3D-array of k clusters with assigned datapoints
208 clusters = np.array([data_norm[np.where(assignments == j)] for j in range(k)])
209
210 # print number of samples in each cluster
211 for j in range(k):
212     print("Number of samples in cluster %d: %d" %(j, len(clusters[j])))
```

H q7_pca.py

```python
#!/usr/bin/env python
#
# MAD 2020-21, Exam number: 23
# Q7 (PCA)
#
# Code mainly developed or provided as part of A5

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

from q7_clustering import centroids, assignments, data_norm as features

def __PCA(data):
    """
    From A5
    """
    data_cent = data - np.mean(data)
    Sigma = np.cov(data_cent.T)
    PCevals, PCevecs = np.linalg.eigh(Sigma)
    PCevals = np.flipud(PCevals) # vertical flip
    PCevecs = np.flip(PCevecs, axis=1) # horisontal flip
    return PCevals, PCevecs

def __transformData(features, PCevecs):
    """
    From A5
    """
    return np.dot(features,  PCevecs[:, 0:2])

PCevals, PCevecs = __PCA(features)

# Convert data to two dimemsions using PCA
features2D = __transformData(features, PCevecs)
centroids2D = __transformData(centroids, PCevecs)

def __visualizeLabels(features, centroids, referenceLabels):
    """
    From A5 (modified)
    """

    plt.figure()
    cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
    cmap_bold  = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])
    y = referenceLabels

    plt.scatter(features[:, 0], features[:, 1], c = y, cmap = cmap_bold)
    plt.scatter(centroids[:, 0], centroids[:,1], c = 'black', s=100)
    plt.xlim(features[:, 0].min() - 0.1, features[:, 0].max() + 0.1)
```

```
50      plt.ylim(features[:, 1].min() - 0.1, features[:, 1].max() + 0.1)
51      plt.show()
52
53  __visualizeLabels(features2D, centroids2D, assignments)
```