Statistical Machine Learning Homework 3

Patrick Nowak
Rinor Cakaj

Sommersemester 2020
July 12, 2020
Sheet 3

Task 3.1: Linear Regression

In this exercise, we will implement various kinds of linear regression using the given data. For all subtasks, we assume that the data $(x_i, y_i)_{i\in\{1,\dots,n\}}$ (n is the number of points) is identically and independently distributed according to

$$y_i = \Phi(x_i)^T w + \epsilon_i,$$

where

$$\epsilon_i \sim \mathcal{N}(0, \sigma^2),$$

hence ϵ_i is normally distributed with mean 0 and variance σ^2. The function $\Phi : \mathbb{R}^1 \to \mathbb{R}^n$ is a feature transformation such that

$$y \sim \mathcal{N}(\Phi(X)^T w, \sigma^2 I).$$

If no basis function is stated explicitely we use the data as is $\Phi(x) = x$.

3.1a)

In the following task we implement linear ridge regression using linear features, i.e. the data itself. We include an additional input dimension to represent a bias term and we use the ridge coefficient $\lambda = 0.01$.

1. It turns out that the classical linear regression is subject to overfitting if the number of attributes is relatively large compared to the number of training points. Christopher M. Bishop mentions in his book "Pattern Recognition and Machine Learning" that one technique that is often used to control the overfitting phenomenon in such cases is that of regularization, which involves adding a penalty term to the error function in order to discourage the coefficients from reaching large values. If we consider ℓ_2-regularization we get the so-called ridge-regression model

$$w = \arg\min_w \frac{1}{2}||X^T w - y||^2 + \frac{\lambda}{2}||w||^2.$$

   The parameter λ is then the ridge coefficient. If λ is big, then we put more emphasis on getting small w which means that we avoid overfitting.
   Furthermore there exist a statistic argument for ridge regression. If we assume that

$$P[y|x, w] \sim \mathcal{N}(y|w^T\phi(x), \sigma^2) \text{ and } w \sim \mathcal{N}(0|\tau^2) \text{ independently}$$

   we can show that maximizing the likehood function is equivalent to minimize

$$w = \arg\min_w \frac{1}{2}||\hat{X}^T w - y||^2 + \frac{\lambda}{2}||w||^2.$$

   where $\hat{X}$ is our transformed data.

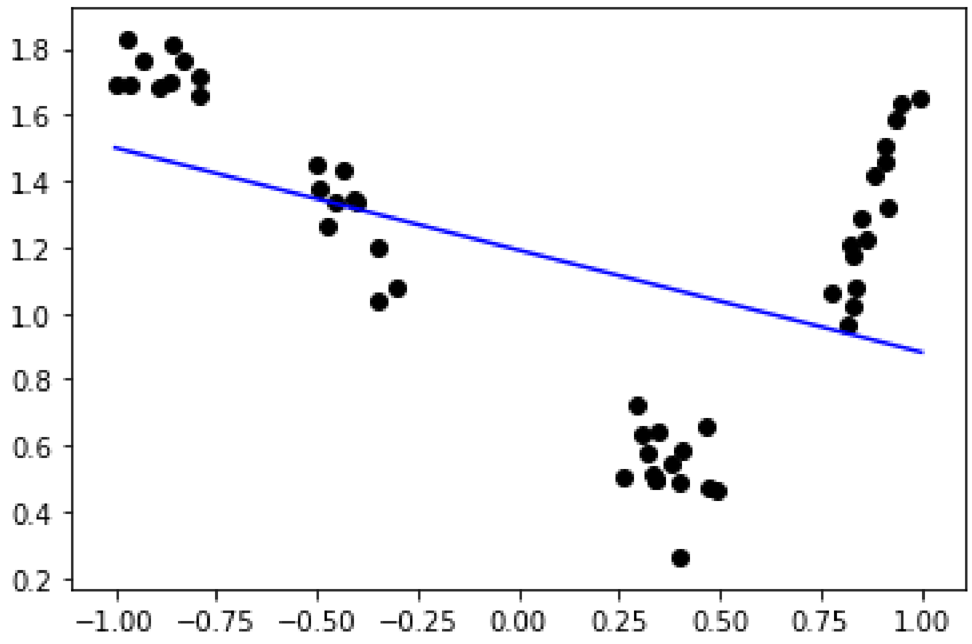
2. To derive the optimal model parameters by minimizing the squared loss function we use the solution formula on page 40.

$$w = (\hat{X}\hat{X}^T + \lambda I)^{-1}\hat{X}y$$

After implementation in python we get $w = \begin{pmatrix}-0.30946806 & 1.19109478\end{pmatrix}$.

3. The root mean squared error of the train data is 0.41217801567361084. The root mean squared error of the test data is 0.38428816992597875.

4. The following plot shows the training data as black dots and the predicted function as a blue line.



Snippets of the code for 1a:

```
def preprocess_X(X):
    ones_vector = np.ones(len(X))
    X_new = np.column_stack((X, ones_vector))
    return X_new.T

def lin_ridge_regression(X_train, y_train, lam):
    X_train_new = preprocess_X(X_train)
    x_x_T = np.matmul(X_train_new, X_train_new.T)
    I = np.eye(len(x_x_T))
    lambda_I = lam * I
    invers = np.linalg.inv(x_x_T + lambda_I)
    w = np.matmul(np.matmul(invers, X_train_new), y_train)
    return w

def solution_value(X, w):
    X_new = preprocess_X(X)
    return np.matmul(X_new.T,w)

def root_mean_squared_error(X_test ,w ,y_test):
    y_pred = solution_value(X_test, w)
    difference = np.linalg.norm(y_pred - y_test)/np.sqrt(len(y_pred))
    return difference
```

```
 if __name__ == "__main__":
     # 1a
     w = lin_ridge_regression(lin_reg_train[:,0], lin_reg_train[:,1], 0.01)
     print("w linear ridge regression: ", w)
     y_pred = solution_value(lin_reg_test[:,0], w)

     difference_1a_train = root_mean_squared_error(lin_reg_train[:,0], w, lin_reg_train[:
     print("Root mean squared error Train a", difference_1a_train)


     difference_1a_test = root_mean_squared_error(lin_reg_test[:,0], w, lin_reg_test[:,1]
     print("Root mean squared error Test a", difference_1a_test)

     #plot
     plt.scatter(lin_reg_train[:,0], lin_reg_train[:,1], c = "black")

     x = np.linspace(-1,1,100) # 100 linearly spaced numbers
     y_pred = solution_value(x,w)

     plt.plot(x,y_pred, c="blue")
     plt.show()
```

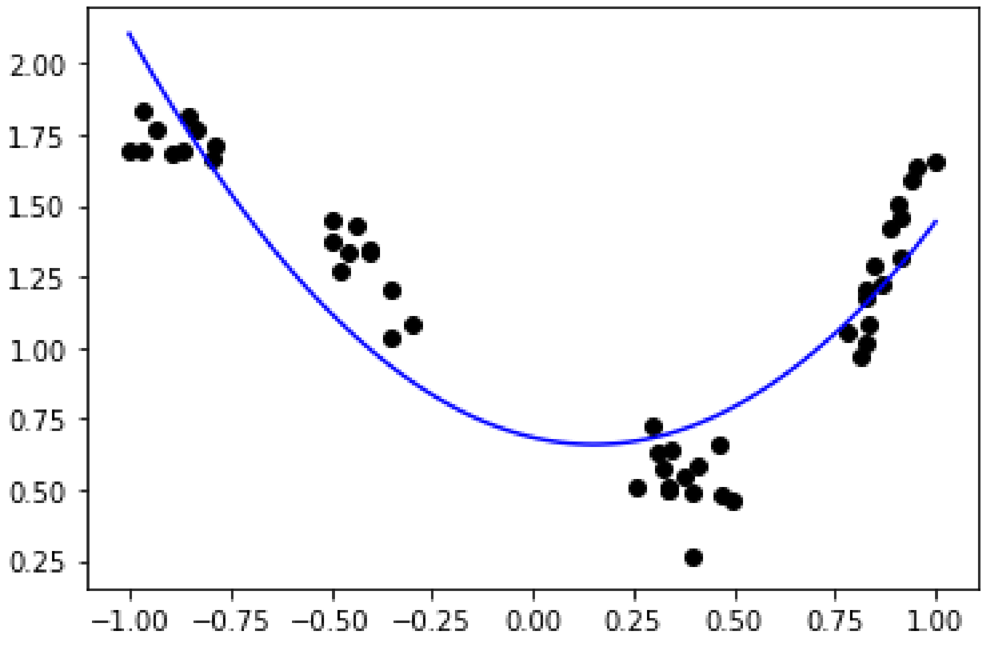
3.1b)

In this subtask we will implement linear ridge regression using a polynomial feature projection. We include an additional input dimension to represent a bias term and use the ridge coefficient $\lambda = 0.01$.

1. Polynomials of degree 2:

   a) The root mean squared error of the training data is 0.21201447265968612. The root mean squared error of the testing data is 0.21687242714148733.

   b) Single plot showing the training data as black dots and the predicted function as blue line:

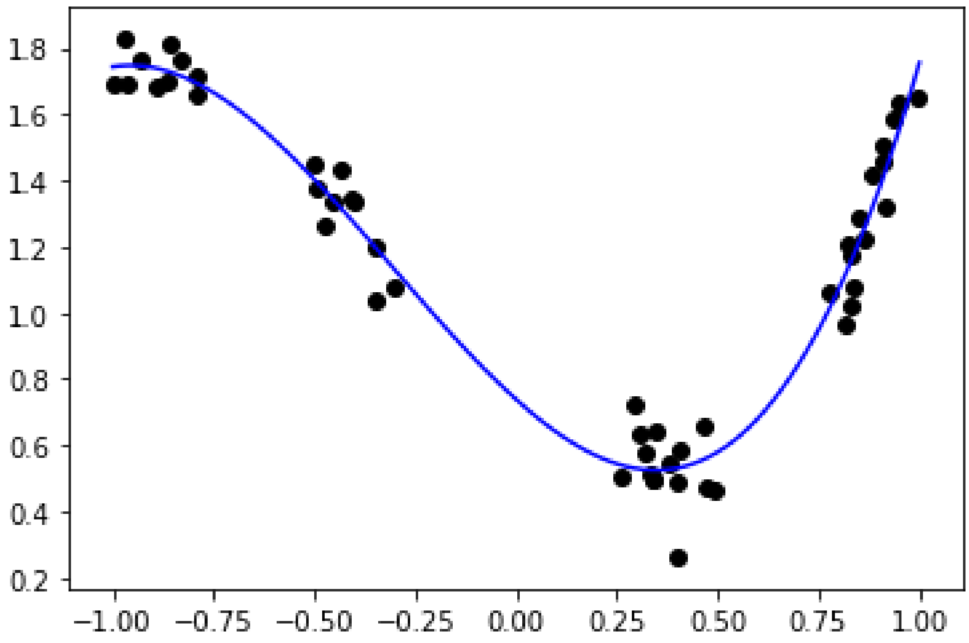


c) One can combine linear regression with a nonlinear feature mapping to fit nonlinear function, i.e. we predict $w^T \phi(x)$ for some feature mapping $\phi : X \to \mathbb{R}^t$. The objective function that we want to fit stays $w^T x$. Hence the function that we want to fit is a nonlinear function of x, but it is a linear function of the coefficients w. Hence this method is called *linear* regression.

2. Polynomials of degree 3:

a) The root mean squared error of the training data is 0.0870682129548175. The root mean squared error of the testing data is 0.10835803719738038.

b) Single plot showing the training data as black dots and the predicted function as blue line:

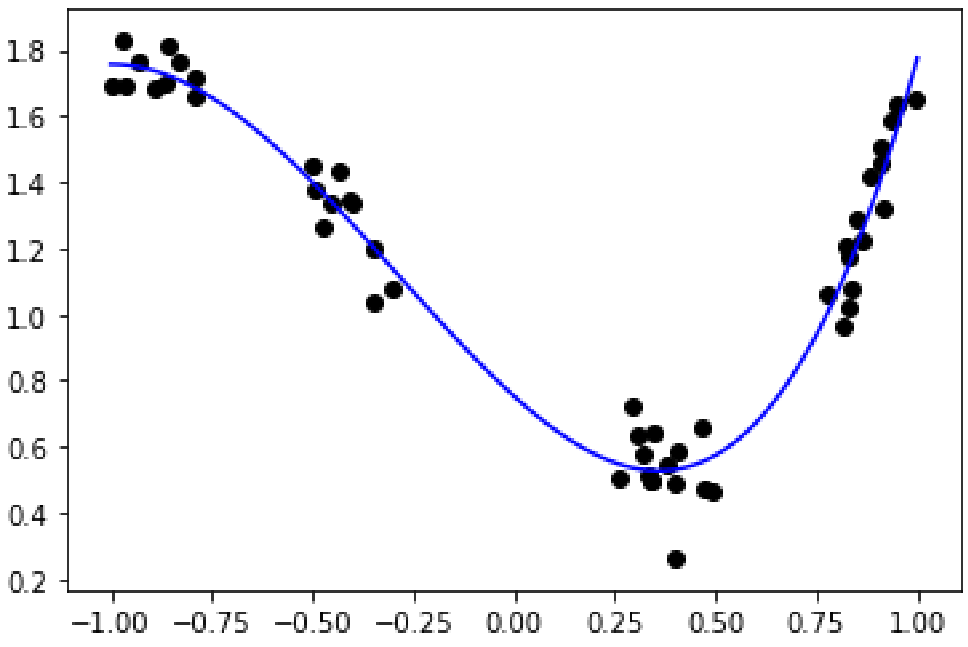


c) See answer for degree 2.

3. Polynomials of degree 4:

a) The root mean squared error of the training data is 0.08701261306638179. The root mean squared error of the testing data is 0.10666239820964699.

b) Single plot showing the training data as black dots and the predicted function as blue line:



c) See answer for degree 2.

Snippets of the code for 1b:

```
def preprocess_X(X):
    ones_vector = np.ones(len(X))
    X_new = np.column_stack((X, ones_vector))
    return X_new.T

def lin_ridge_regression(X_train, y_train, lam):
    X_train_new = preprocess_X(X_train)
    x_x_T = np.matmul(X_train_new, X_train_new.T)
    I = np.eye(len(x_x_T))
    lambda_I = lam * I
    invers = np.linalg.inv(x_x_T + lambda_I)
    w = np.matmul(np.matmul(invers, X_train_new), y_train)
    return w

def solution_value(X, w):
    X_new = preprocess_X(X)
    return np.matmul(X_new.T,w)

def root_mean_squared_error(X_test ,w ,y_test):
    y_pred = solution_value(X_test, w)
    difference = np.linalg.norm(y_pred - y_test)/np.sqrt(len(y_pred))
    return difference

def stack(X_1, degree):
    X = X_1
    for i in range(2,degree+1):
        X = np.column_stack((X,np.power(X_1,i)))
    return X

#1b
    X_train = lin_reg_train[:,0]
```

```
y_train = lin_reg_train[:,1]
X_test = lin_reg_test[:,0]
y_test = lin_reg_test[:,1]
degree = 2

X_train_new = stack(X_train, degree)
w1 = lin_ridge_regression(X_train_new, y_train, 0.01)
print("w linear ridge regression using a polynomial feature projection: ", w1)
difference_1b_train = root_mean_squared_error(X_train_new, w1, y_train)
print("Root mean squared error Train b ", difference_1b_train)

X_test_new = stack(X_test, degree)
difference_1b_test = root_mean_squared_error(X_test_new, w1, y_test)
print("Root squared mean error Test b ", difference_1b_test)

x = np.linspace(-1,1,100) # 100 linearly spaced numbers
X_1 = stack(x, degree)
y_pred = solution_value(X_1, w1)

plt.scatter(lin_reg_train[:,0], lin_reg_train[:,1], c = "black")
plt.plot(x,y_pred, c="blue")
plt.show()
```

3.1c)

In this subtask we implement 5-fold cross-validation to select the optimal degree for our polynomial degression.

1. Polynomial degree: 2
   a) Average train RMSE among all folds: 0.20943990135161356
   b) Average validation RMSE among all folds: 0.2248850559783977
   c) Average test RMSE among all folds: 0.21835094011192718
2. Polynomial degree: 3
   a) Average train RMSE among all folds: 0.08620813857069495
   b) Average validation RMSE among all folds: 0.09271100111785263
   c) Average test RMSE among all folds: 0.10927671570049995
3. Polynomial degree: 4
   a) Average train RMSE among all folds: 0.08547251620354353
   b) Average validation RMSE among all folds: 0.09841566883354039
   c) Average test RMSE among all folds: 0.10867173876433564
4. In the plots where the training data is shown as black dots one can cluster the points into 4 groups. At the beginning I thought that maybe the provided data is split in such a way that one group is in the validation set and the other groups are in the training set. In this situation, the cross-validation would perform very bad.
   Since the training data is not ordered along the x-axis, the case described above did not happen. We excluded points from the training set that came from all groups.
5. The polynomial degree 3 should be chosen, since the complexity of solving the system for polynomial degree 3 is less than for polynomial degree 4 and the polynomial with degree 4 does not perform much better than the polynomial with degree 3. Furthermore the polynomial of degree 3 performs better on the validation set then polynomial of degree 4.

Code snippets for 1c):

```
def preprocess_X(X):
    ones_vector = np.ones(len(X))
    X_new = np.column_stack((X, ones_vector))
    return X_new.T

def lin_ridge_regression(X_train, y_train, lam):
    X_train_new = preprocess_X(X_train)
    x_x_T = np.matmul(X_train_new, X_train_new.T)
    I = np.eye(len(x_x_T))
    lambda_I = lam * I
    invers = np.linalg.inv(x_x_T + lambda_I)
    w = np.matmul(np.matmul(invers, X_train_new), y_train)
    return w

def solution_value(X, w):
    X_new = preprocess_X(X)
    return np.matmul(X_new.T,w)


def root_mean_squared_error(X_test ,w ,y_test):
    y_pred = solution_value(X_test, w)
    difference = np.linalg.norm(y_pred - y_test)/np.sqrt(len(y_pred))
    return difference

def stack(X_1, degree):
    X = X_1
    for i in range(2,degree+1):
        X = np.column_stack((X,np.power(X_1,i)))
    return X


def five_fold_cross_validation(train_data, test_data):
    X_1 = train_data[0:10]
    X_2 = train_data[10:20]
    X_3 = train_data[20:30]
    X_4 = train_data[30:40]
    X_5 = train_data[40:50]


    # Use subsets 1 - 4 to train your model with polynomial features of
    degrees 2, 3 and 4.
    degrees = [2,3,4]
    subset_5 = [np.concatenate((X_1,X_2,X_3,X_4), axis = 0), X_5]
    subset_4 = [np.concatenate((X_1,X_2,X_3,X_5), axis = 0), X_4]
    subset_1 = [np.concatenate((X_2,X_3,X_4,X_5), axis = 0), X_1]
    subset_2 = [np.concatenate((X_3,X_4,X_5,X_1), axis = 0), X_2]
    subset_3 = [np.concatenate((X_4,X_5,X_1,X_2), axis = 0), X_3]




    subsets = [subset_1, subset_2, subset_3, subset_4, subset_5]
```

```
for degree in degrees:
    RMSE_test = 0
    cross_RMSE_train = 0
    cross_RMSE_test = 0
    for subset in subsets:
        # define train and test set
        cross_validation_X_train = subset[0][:,0]
        cross_validation_y_train = subset[0][:,1]
        cross_validation_X_test = subset[1][:,0]
        cross_validation_y_test = subset[1][:,1]
        lam = 0.01


        cross_validation_X_train_new = stack(cross_validation_X_train, degree)

        w = lin_ridge_regression(cross_validation_X_train_new, cross_
        validation_y_train, lam)

        # RMSE for cross_validation
        cross_RMSE_train = cross_RMSE_train + root_mean_squared_error(cross_
        validation_X_train_new, w, cross_validation_y_train)

        cross_validation_X_test_new = stack(cross_validation_X_test, degree)

        cross_RMSE_test = cross_RMSE_test + root_mean_squared_error(cross_
        validation_X_test_new, w, cross_validation_y_test)

        #RMSE for test data
        X_test = test_data[:,0]
        y_test = test_data[:,1]

        X_test_new = stack(X_test, degree)

        RMSE_test = RMSE_test + root_mean_squared_error(X_test_new, w, y_test)
    print("degree: ", degree, "Average Train RMSE: ", cross_RMSE_train/5,
    "Average Validation RMSE:", cross_RMSE_test/5, "Average Test RMSE: ", RMSE_test/5)
```

3.1d)

In this subtask we implement Bayesian linear ridge regression, assuming that w follows a multivariate Gaussian distribution, such that

$$w \sim \mathcal{N}(\mu_0, \Lambda_0^{-1})$$

where ridge regression dictates $\mu_0 = 0$ and $\Lambda_0 = \lambda I$.
We assume $\sigma = 0.1$ and $\lambda = 0.01$ and include an additional input dimension to represent a bias term. We use all of the provided training data for a single Bayesian update.

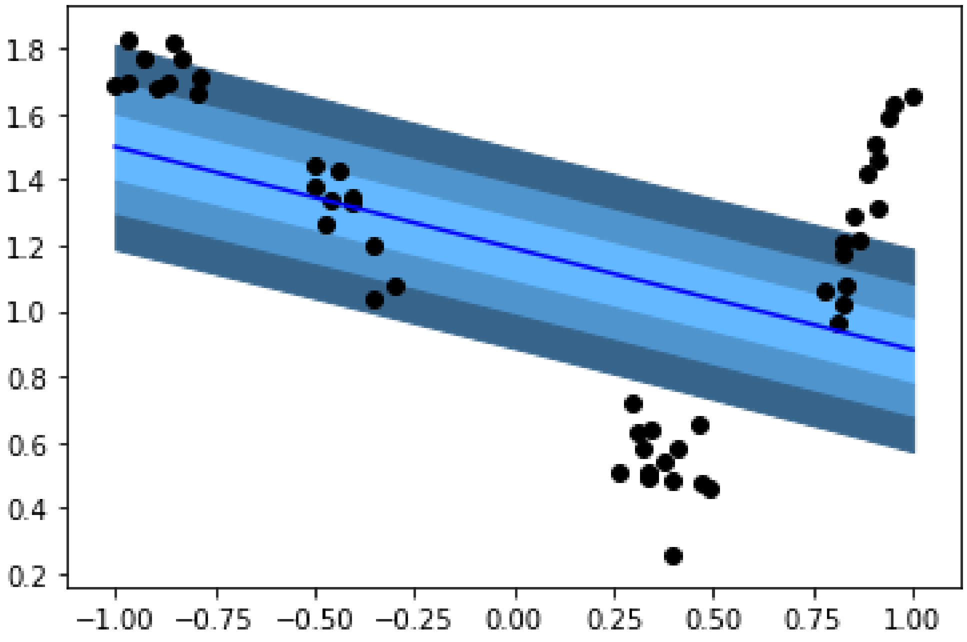
1. The posterior distribution of the model parameters $p(w|X, y)$ is

$$\begin{aligned} p(w|X,y) &= \mathcal{N}(\mu_n, \Lambda_n^{-1}) \\ \mu_n &= \Lambda_n^{-1}(\sigma^{-2}\Phi^T y) \\ \Lambda_n &= \Lambda_0 + \sigma^{-2}\Phi^T\Phi \end{aligned}$$

2. Let X_* be a batch of predictions. Then the predictive distribution is

$$p(y^*|X_*, X, y) = \frac{1}{\sqrt{(2*\pi)^p \det(\Sigma)}} \exp(-\frac{1}{2}(y^* - \mu)^T \Sigma^{-1}(y^* - \mu))$$
$$\mu(X_*) = \phi(X_*)(\frac{\alpha}{\beta} I + \Phi\Phi^T)^{-1}\Phi^T y$$
$$\Sigma = \frac{1}{\beta} I + \phi^T(X_*)(\alpha I + \beta\Phi\Phi^T)^{-1}\phi(X_*)$$

3. The RMSE of the train data under our Bayesian model is: 0.41217792591659724. The RMSE of the test data under our Bayesian model is: 0.38434085452132943.
4. The average log-likehood of the train data under our Bayesian model is: -6.834699569913453. The average log-likehood of the test data under our Bayesian model is: -5.774748828572731.
5. The following plot shows the training data as black dots, the mean of the predictive distribution as blue line and 1,2 and 3 standard deviations of the predictive distributions in shades of blue



6. The main difference between linear regression and Bayesian linear regression is that in linear regression we get an explicit w such that the linear mapping $x^T w$ describes our data well. In the case of Bayesian linear regression we get a predictive distribution which is Gaussian. The distribution depends on the noise of the data and the Gaussian prior of w. Furthermore the Bayesian treatment of linear regression will avoid over-fitting problem of the maximum-likehood and will also lead to automatic methods of determining model complexity using only the training data. (Bishop)

Snippets of code for 1d):

```
def preprocess_X(X):
    ones_vector = np.ones(len(X))
    X_new = np.column_stack((X, ones_vector))
    return X_new.T

def bayesian_linear_ridge_regression_mu(X,x,y):
    alpha = 0.01
    beta = 100
    X_new = preprocess_X(X)
    x_x_T = np.matmul(X_new, X_new.T)
    I = np.eye(len(x_x_T))
    lambda_I = (alpha/beta) * I
    invers = np.linalg.inv(x_x_T + lambda_I)
    w = np.matmul(np.matmul(invers, X_new), y_train)
    if type(x) != int:
        x_new = preprocess_X(x)
    else:
        x_new = np.asarray([x, 1])
    mu = np.matmul(x_new.T,w)
    return mu

def rmse_bayesian(X_train,y_train, data_set_x, data_set_y):
    y_pred = bayesian_linear_ridge_regression_mu(X_train, data_set_x, y_train)
    difference = np.linalg.norm(y_pred - data_set_y)/np.sqrt(len(y_pred))
    return difference

def log_likehood_bayesian(X_train, y_train, data_set_x, data_set_y, g=0, alpha=0.01,
beta=100):

    average_log_likehood = 0

    mu = bayesian_linear_ridge_regression_mu(X_train, data_set_x, y_train)

    dev = np.zeros(len(mu))

    for i in range(len(data_set_x)):
        X_new = preprocess_X(X_train)
        x_x_T = np.matmul(X_new, X_new.T)
        I = np.eye(len(x_x_T))
        alpha_I = alpha * I
        beta_x_x_T = beta * x_x_T
        invers = np.linalg.inv(alpha_I + beta_x_x_T)
        if type(data_set_x[i]) == np.float64:
            data_set_x_new = np.append(data_set_x[i],1)
        else:
            data_set_x_new = preprocess_X(data_set_x[i].T)
        sigma_pre = np.matmul(np.matmul(data_set_x_new.T, invers),data_set_x_new)
        sigma = 1/beta + sigma_pre

        dev[i] = np.sqrt(sigma)

        # print("mu:", mu)
        # print("sigma", sigma)
        # print("x", data_set_x[i])
        # print("y", data_set_y[i])

        average_log_likehood = average_log_likehood + gaussian_density_function(mu[i],
        sigma, data_set_y[i])


    if g==1:
        plt.plot(data_set_x, mu, c = "blue")
        plt.fill_between(data_set_x, mu, mu + dev, color='#63B8FF')
        plt.fill_between(data_set_x, mu, mu - dev, color='#63B8FF')
        plt.fill_between(data_set_x, mu + dev, mu + 2*dev, color='#4F94CD')
        plt.fill_between(data_set_x, mu - dev , mu - 2*dev, color='#4F94CD')
        plt.fill_between(data_set_x, mu + 2* dev, mu + 3*dev, color='#36648B')
        plt.fill_between(data_set_x, mu - 2*dev, mu - 3*dev, color='#36648B')

        plt.scatter(lin_reg_train[:,0], lin_reg_train[:,1], c = "black")
        plt.show()



    return average_log_likehood/len(data_set_x)
#1d

    X_train = lin_reg_train[:,0]
    y_train = lin_reg_train[:,1]
    X_test = lin_reg_test[:,0]
    y_test = lin_reg_test[:,1]

    bayesian_linear_ridge_regression_mu(X_train,[1, 2],y_train)

    # Report the RMSE of the train and test data under your Bayesian model
    (use the predictive mean)

    print("RMSE Bayesian_training: ", rmse_bayesian(X_train,y_train, X_train, y_train))

    print("RMSE Bayesian Test: ",rmse_bayesian(X_train, y_train, X_test, y_test))

    # Report the average log-likelihood of the train and test data under your
    Bayesian model.

    print("Log Likehood Train: ",log_likehood_bayesian(X_train, y_train, X_train, y_train))

    print("Log Likehood Test: ",log_likehood_bayesian(X_train, y_train, X_test, y_test))

    x = np.linspace(-1,1,100) # 100 linearly spaced numbers

    log_likehood_bayesian(X_train, y_train, x, y_test, g=1)
```

3.1e)

In the following we implement Bayesian linear ridge regression using linear ridge regression model using squared exponential (SE) features. In other word, we replace our observed data matrix $X \in \mathbb{R}^{nx1}$ by a feature matrix $\Phi \in \mathbb{R}^{nxk}$, where

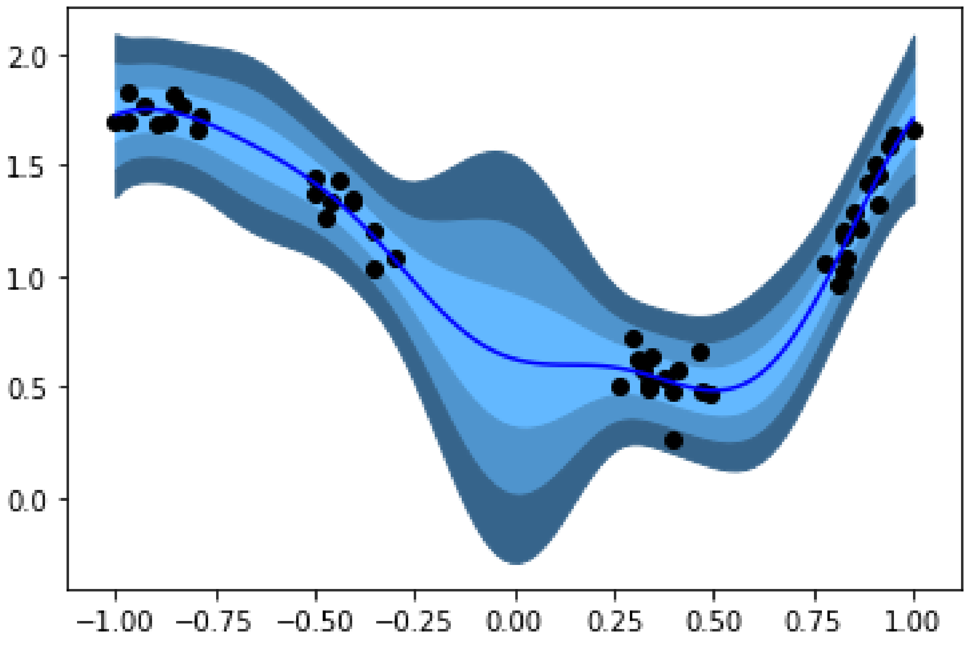
$$\Phi_{ij} = \exp(-\frac{1}{2}\beta(X_i - \alpha_j)^2).$$

Set $k = 20$, $\alpha_j = j * 0.1 - 1$ and $\beta = 10$. We use the ridge coefficient $\lambda = 0.01$ and assume known Gaussian noise with $\sigma = 0.1$. We include an additional input dimension to represent a bias term.

1. The RMSE of the train data under our Bayesian model with SE features is: 0.08160941530998907. The RMSE of the test data under our Bayesian model with SE features is: 0.14341009681107458.

2. The average log-likehood of the train data under our Bayesian model with SE feature is: 1.013494830677245. The average log-likehood of the test data under our Bayesian model with SE feature is: 0.5595070674582804.

3. The following single plot show the training data as black dots, the mean of the predictive distribution as blue line and 1,2 and 3 standard deviations of the predictive distribution in shades of blue.



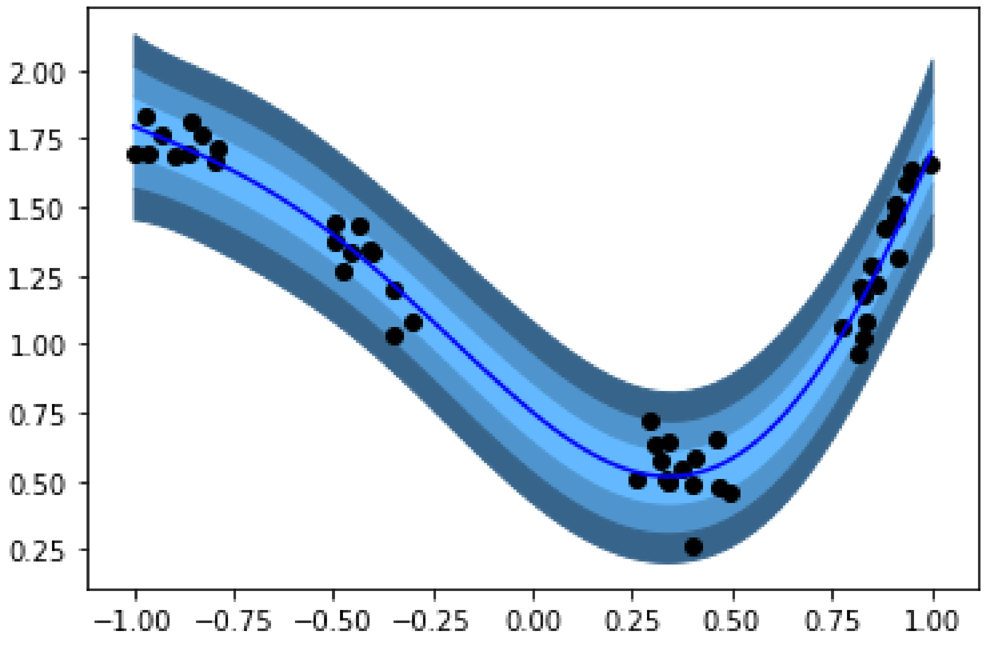
4. From a statisticians point of view SE features can be interpreted as a transformation with Gaussians. α is the mean and β^{-1} is the variance. We take an observation X_i and get for $k = 20$ 20 columns. For each column j we calculate the value of the Gaussian density(not exactly the Gaussian density, since there is missing a term in front of exp) with variance β^{-1} and mean α_j. If α_j for column j is far away from X_i then this entry in the column will not have big effect on our prediction since it is close to zero. That is also the reason why α_j runs from -1 to 1, since all of our data are from this interval.

Code snippets for 1e):

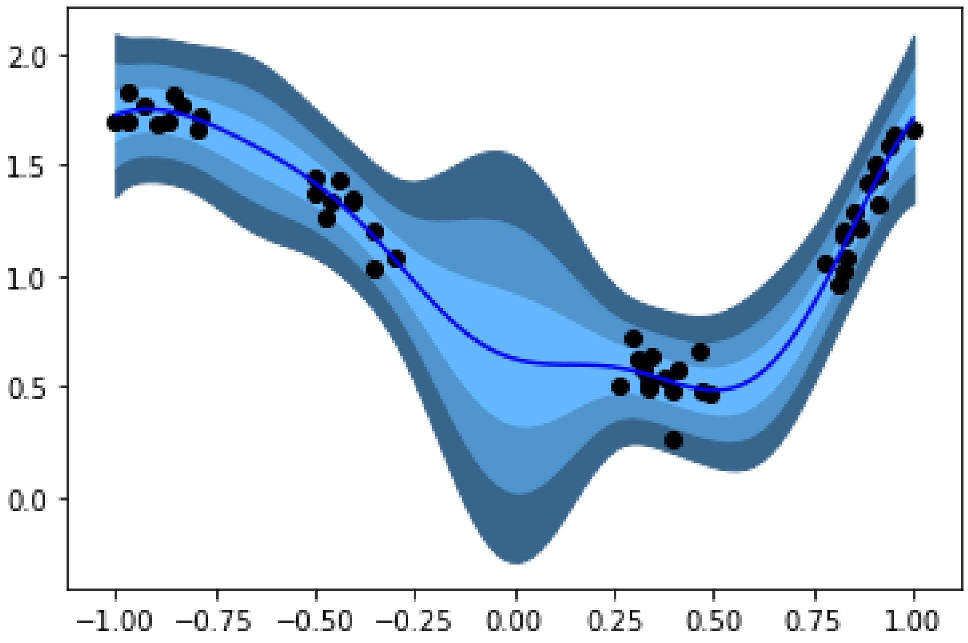
```
def gaussian_density_function(mu, sigma, x):
    c1 = 1 / np.sqrt(2*np.pi * sigma)
    c2 = -0.5 * (x - mu)**2/ sigma
    y = np.log(c1) + c2
    return y

def preprocess_X(X):
    ones_vector = np.ones(len(X))
    X_new = np.column_stack((X, ones_vector))
    return X_new.T

def bayesian_linear_ridge_regression_mu(X,x,y):
    alpha = 0.01
    beta = 100

    X_new = preprocess_X(X)
    x_x_T = np.matmul(X_new, X_new.T)
    I = np.eye(len(x_x_T))
    lambda_I = (alpha/beta) * I
    invers = np.linalg.inv(x_x_T + lambda_I)
    w = np.matmul(np.matmul(invers , X_new), y_train)
    if type(x) != int:
        x_new = preprocess_X(x)
    else:
        x_new = np.asarray([x, 1])
    mu = np.matmul(x_new.T,w)
    return mu

def rmse_bayesian(X_train ,y_train , data_set_x , data_set_y):
    y_pred = bayesian_linear_ridge_regression_mu(X_train , data_set_x , y_train)
    difference = np.linalg.norm(y_pred - data_set_y)/np.sqrt(len(y_pred))
    return difference

def log_likehood_bayesian_SE(X_train , y_train , data_set_x , data_set_y , data_set_x_pre ,
g=0, alpha=0.01, beta=100):

    average_log_likehood = 0

    mu = bayesian_linear_ridge_regression_mu(X_train , data_set_x , y_train)

    dev = np.zeros(len(mu))

    for i in range(len(data_set_x)):
        X_new = preprocess_X(X_train)
        x_x_T = np.matmul(X_new, X_new.T)
        I = np.eye(len(x_x_T))
        alpha_I = alpha * I
        beta_x_x_T = beta * x_x_T
        invers = np.linalg.inv(alpha_I + beta_x_x_T)
        if type(data_set_x[i]) == np.float64 or len(data_set_x[i] == 1):
            data_set_x_new = np.append(data_set_x[i],1)
        else:
            data_set_x_new = preprocess_X(data_set_x[i].T)
        sigma_pre = np.matmul(np.matmul(data_set_x_new.T, invers),data_set_x_new)
        sigma = 1/beta + sigma_pre

        dev[i] = np.sqrt(sigma)

        # print("mu:", mu)
        # print("sigma", sigma)
        # print("x", data_set_x[i])
        # print("y", data_set_y[i])


        average_log_likehood = average_log_likehood + gaussian_density_function(mu[i],
        sigma, data_set_y[i])

    if g == 1:



        plt.plot(data_set_x_pre , mu, c = "blue")
        plt.fill_between(data_set_x_pre , mu, mu + dev, color='#63B8FF')
        plt.fill_between(data_set_x_pre , mu, mu - dev, color='#63B8FF')
        plt.fill_between(data_set_x_pre , mu + dev, mu + 2*dev, color='#4F94CD')
        plt.fill_between(data_set_x_pre , mu - dev , mu - 2*dev, color='#4F94CD')
        plt.fill_between(data_set_x_pre, mu + 2* dev, mu + 3*dev, color='#36648B')
        plt.fill_between(data_set_x_pre, mu - 2*dev, mu - 3*dev, color='#36648B')

        plt.scatter(lin_reg_train[:,0], lin_reg_train[:,1], c = "black")
        plt.show()



    return average_log_likehood/len(data_set_x)

def feature_mapping(X,k, beta):
    PHI_pre = np.zeros((len(X),k))
    for i in range(len(X)):
        for j in range(k):
            alpha_j = (j+1) * 0.1 - 1
            PHI_pre[i,j] = np.exp(-0.5 * beta * (X[i] - alpha_j)**2)
    return PHI_pre


 #1e

    k = 20
    beta = 10
    PHI = feature_mapping(X_train,k, beta) # vorsicht! samples in den zeilen

    PHI_test = feature_mapping(X_test,k, beta)


    # # Report the RMSE of the train and test data under your Bayesian model with SE features

    print("RMSE Bayesian_SE_training: ", rmse_bayesian(PHI ,y_train, PHI, y_train))

    print("RMSE Bayesian_SE Test: ",rmse_bayesian(PHI, y_train, PHI_test, y_test))

    # # Report the average log-likelihood of the train and test data under your Bayesian mod


    print("Average Log Likehood SE Train: ",log_likehood_bayesian_SE(PHI, y_train,
    PHI, y_train, X_train))

    print("Average Log Likehood SE Test: ",log_likehood_bayesian_SE(PHI, y_train,
    PHI_test, y_test, X_test))

    # Real Plot

    x = np.linspace(-1,1,100) # 100 linearly spaced numbers

    X = feature_mapping(x,k, beta)

    g=1

    log_likehood_bayesian_SE(PHI, y_train, X, y_test, x, g)
```

3.1f)

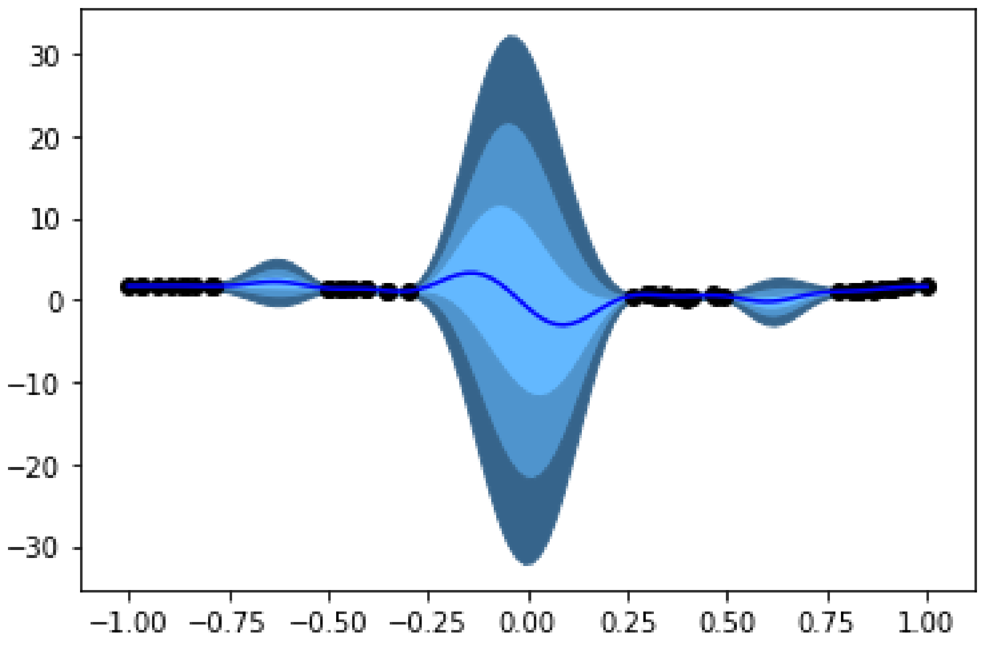
1. The difference between the marginal likelihood $p(y|X)$ and the likelihood $p(y|X,w)$ is that the marginal likelihood does not depend on w. We do not have to assume any distribution on w.
2. For each β, report RMSE, average log-likehood of the train and test data and the log-marginal likelihood.
   a) $\beta = 1$
      i. RMSE of the train data: 0.09057345725244549. RMSE of the test data: 0.10867908746213598
      ii. Average log-likelihood of the train data: 0.9608029250737622. Average log-likelihood of the test data: 0.8002394451718441
      iii. log-marginal likelihood on the train data: 24.045413450553077
   b) $\beta = 10$
      i. RMSE of the train data: 0.08160941530998907. RMSE of the test data: 0.14341009681107458
      ii. Average log-likelihood of the train data: 1.013494830677245. Average log-likelihood of the test data: 0.5595070674582804
      iii. log-marginal likelihood on the train data: 11.632969636668967
   c) $\beta = 100$
      i. RMSE of the train data: 0.07011271637816664. RMSE of the test data: 1.0488486795792595
      ii. Average log-likelihood of the train data: 1.041261156525514. Average log-likelihood of the test data: -0.5288965833585361
      iii. log-marginal likelihood on the train data: -21.90536066926914
3. For each β, include a single plot that shows the training data as black dots, the mean of the predictive distribution as blue line and 1,2,3 standard deviations of the predictive distribution in shades of blue.
   a) $\beta = 1$



   b) $\beta = 10$



c) $\beta = 100$



4. Using the log-marginal likelihood as a score function $\beta = 1$ is the best value, since we wan't to maximize the log-marginal likelihood. We wan't to maximize the log-marginal likelihood, since we want that the probability of predicting y under the assumption that X is given is high.
5. We observe that the average train log-likelihood is for $\beta = 100$ better then for $\beta = 1$ but the log-marginal likelihood is for $\beta = 1$ is much greater then for $\beta = 100$. If we look now on the average test log-likelihood, one can see that the average test log-likelihood is better for $\beta = 1$ then for $\beta = 100$. Hence the log-marginal likelihood is a better score function for the training data to perform grid search, since the β we decide will perform in general better for the test set.

Code snippets for 1f).

```
#1f
    beta_all = [1,10,100]
    for beta in beta_all:
        g=0 # paramter for plotting
        k = 20
        print("beta: ", beta)
        PHI = feature_mapping(X_train ,k, beta) # vorsicht! samples in den zeilen

        PHI_test = feature_mapping(X_test ,k, beta)


        #Report the RMSE of the train and test data under your Bayesian model with SE feature

        print("RMSE 1f train: ", rmse_bayesian(PHI ,y_train , PHI, y_train))

        print("RMSE 1f test: ",rmse_bayesian(PHI, y_train , PHI_test , y_test))

        #Report the average log−likelihood of the train and test data under your Bayesian mo


        print("Average Log likehood train: ",log_likehood_bayesian_SE(PHI, y_train ,
        PHI, y_train , X_train))

        print("Average Log likehood test: ",log_likehood_bayesian_SE(PHI, y_train ,
        PHI_test , y_test , X_test))

        # Report log−marginal likehood

        print("log_marginal_likehood train:", log_marginal_likelihood(PHI, y_train))


        x = np.linspace(−1,1,100) # 100 linearly spaced numbers

        X = feature_mapping(x,k, beta)

        g=1

        log_likehood_bayesian_SE(PHI, y_train , X, y_test , x, g)
        plt.show()
```

Task 2: Linear Classification

2a)

In the following we focus on the case of two classes C_1 and C_2 to keep the arguments simple. Our main goal is to compute

$$p(C_1|x) \text{ and } p(C_2|x)$$

such that we can decide to which class a data point x belongs. Hence we classify x into the class with the larger posterior probability. A discriminative model (e.g. logistic regression) tries to model these posterior probability directly. A generative model uses the Bayes rule to compute the posterior distribution after modelling first the joint probability distributions $p(x, C_1)$ and $p(x, C_2)$. We get:

$$\begin{aligned} p(C_i|x) &= \frac{p(x|C_i)p(C_i)}{p(x)} && (1) \\ &= \frac{p(x|C_i)p(C_i)}{p(x|C_1)p(C_1) + p(x|C_2)p(C_2)} && (2) \\ &= \frac{p(x, C_i)}{p(x, C_1) + p(x, C_2)} && (3) \end{aligned}$$

The generative approach gives more information about the distribution such that we can sample new data points from the modelled distribution. This is not possible for the discriminative approach.
Many people argue that discriminative approaches are better, because we do not need to solve a more general problem in this case. We can directly compute the posterior probability. Furthermore we have many tricks to improve the discriminative approaches in practice. E.g. for logistic regression we can shrink parameters or impose margin constraints. Ng points out, that in theoretical settings we observe two phenomena. Logistic regression always has a lower asymptotic error than naive Bayes. But the error of naive Bayes decreases faster in the beginning. Hence if we have only few data points it may be better to use naive Bayes instead of logistic regression. However, after a certain amount of data, the logistic error will become lower and also stay lower than the error of naive Bayes. Information from [Ng 2001].

2b)

In this exercise we use Fisher's Linear Discriminant to separate 3 classes. After observing the data we decided to use pairwise Fisher's Linear Discriminant for 2 classes ('one-vs-one'). That means that we separate C_1 and C_2 by L_{12}, then separate C_1 and C_3 by L_{13} and finally separate C_2 and C_3 by L_{23}. L_{ij} is the Fisher's Linear Discriminant Algorithm for classes C_i and C_j. More precisely, we first used Fisher's discriminant approach to project each point to a one-dimensional subspace and then we used Bayesian Decision Theory (assuming a Gaussian distribution after projecting) to decide for C_i or C_j. By majority vote on the three results, we then assigned x to a class.
Out of 137 points in total we misclassified 19. One point from class 1 (red) was misclassified as being class 2 (yellow). 11 points from class 3 (green) were misclassified as being class 2 (yellow). 7 points from class 2 (yellow) were misclassified as being class 3 (green). If we look at the plot we can easily see that it is very hard to separate classes 2 and 3 with a straight line. Further, red and yellow resp. red and green can be separated easily.
In our example this method worked well but the intersection of the three linear classifiers will produce a triangle somewhere in the middle of the plot. This will cause problems if we want to classify points in it as our majority vote will not work there. But none of the points of the data set lay in this triangle.
Code Snippets for Task 2:

```
def biased_ml_estimate(X):
    # mu
    mu = 0
    n = len(X)
    for i in range(n):
```

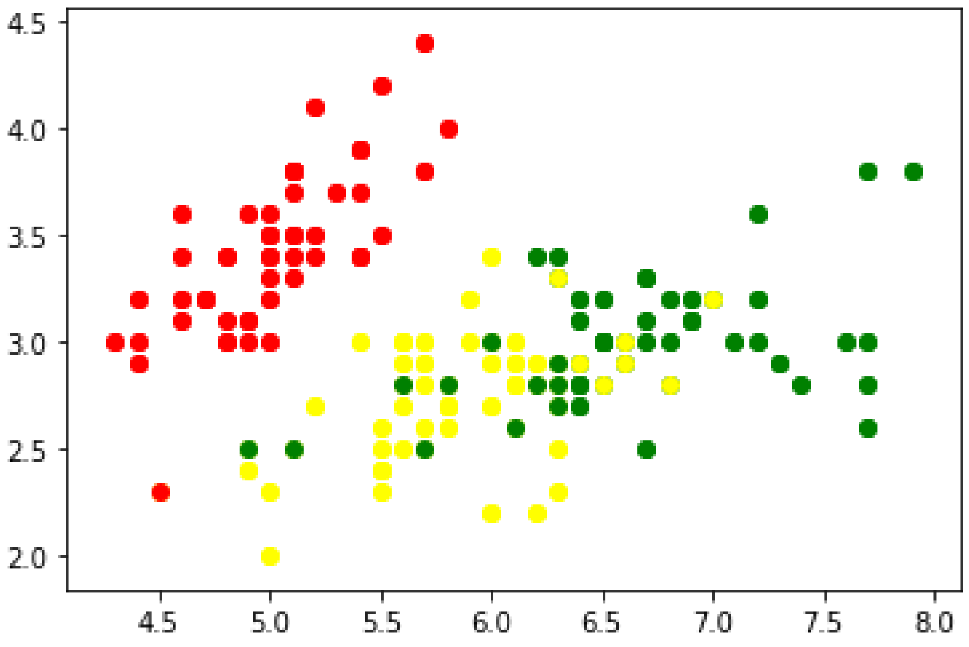


Figure 1: The plot of the three classes with their original labels. Red: Class 1, Yellow: Class 2, Green: Class 3.

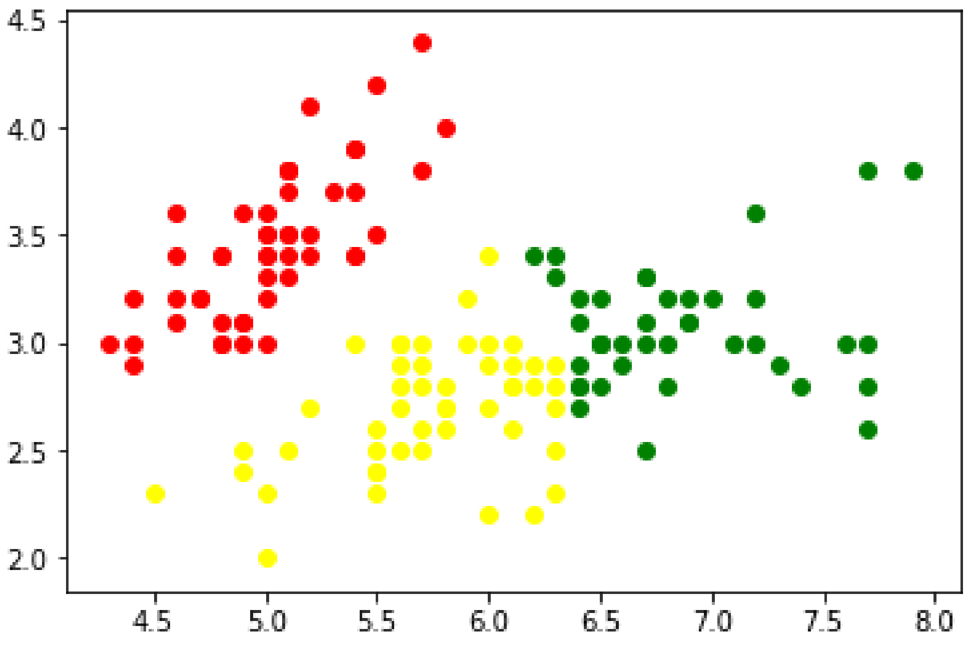


Figure 2: The plot of the three classes with the labels coming from L_{12}, L_{13} and L_{23}. Red: Class 1, Yellow: Class 2, Green: Class 3.

```
        mu = mu + X[i]
    mu = mu / n

    variance = 0
    for p in range(n):
        variance = variance + (X[p] - mu)**2
    variance = variance / (n-1) #for biased estimate divide by n
    return mu, variance

def gaussian_density_function(mu, sigma, x):
    c1 = 1 / np.sqrt(2*np.pi * sigma)
    c2 = np.exp(-0.5 * (x - mu)**2/ sigma)
    y = c1 * c2
    return y

def compute_prior_probabilities(c1,c2):
    length_c1 = len(c1)
    length_c2 = len(c2)

    prob_c1 = length_c1 / (length_c1 + length_c2)
    prob_c2 = length_c2 / (length_c1 + length_c2)

    return(prob_c1, prob_c2)

def S_W_inverse(C_1, C_2):
    n_1 = len(C_1)
    n_2 = len(C_2)

    m_1 = np.mean(C_1, axis=0)
    m_2 = np.mean(C_2, axis=0)


    for i in range(n_1):
        if i == 0:
            sum_1 = np.outer(C_1[i] - m_1,C_1[i] - m_1)
        else:
            sum_1 = np.add(sum_1, np.outer(C_1[i] - m_1,C_1[i] - m_1))

    for i in range(n_2):
        if i == 0:
            sum_2 = np.outer(C_2[i] - m_2,C_2[i] - m_2)
        else:
            sum_2 = np.add(sum_2, np.outer(C_2[i] - m_2,C_2[i] - m_2))
    invers = np.linalg.inv(np.add(sum_1, sum_2))

    return invers

def fisher_linear_discriminant(C_1,C_2):
    S_W_inv = S_W_inverse(C_1, C_2)

    diff = np.mean(C_1, axis=0)-np.mean(C_2, axis=0)

    w = np.matmul(S_W_inv, diff)

    proj_C_1 = np.matmul(C_1, w)
    proj_C_2 = np.matmul(C_2, w)

    prob_c_1, prob_c_2 = compute_prior_probabilities(proj_C_1, proj_C_2)

    mu_1, variance_1 = biased_ml_estimate(proj_C_1)

    mu_2, variance_2 = biased_ml_estimate(proj_C_2)

    return w, mu_1, variance_1, mu_2, variance_2, prob_c_1, prob_c_2

def decision_function(w, mu_1, variance_1, mu_2, variance_2, prob_C_1, prob_C_2, x):
    proj_x = np.matmul(w,x)
    gauss_1 = gaussian_density_function(mu_1, variance_1, proj_x)
    gauss_2 = gaussian_density_function(mu_2, variance_2, proj_x)

    if (gauss_1/gauss_2) > (prob_C_2/prob_C_1):
        print("Point is classified in Class 1:", x)
    else:
        print("Point is classified in Class 2:", x)



if __name__ == "__main__":
    # 2a
    #Class 1 / Class 2
    onetwo_w, onetwo_mu_1, onetwo_variance_1, onetwo_mu_2, onetwo_variance_2, onetwo_prob_C_1
    print("Class 1 / Class 2 - w:",onetwo_w)

    #Class 2 / Class 3
    twothree_w, twothree_mu_2, twothree_variance_2, twothree_mu_3, twothree_variance_3, twoth
    print("Class 2 / Class 3 - w:",twothree_w)

    #Class 1 / Class 3
    onethree_w, onethree_mu_1, onethree_variance_1, onethree_mu_3, onethree_variance_3, oneth
    print("Class 1 / Class 3 - w:",onethree_w)

    counter = 0

    for x in ldaData:
        i = 0
        c1 = 0
        c2 = 0
        c3 = 0

        # 12
        proj_x = np.matmul(onetwo_w,x)
        gauss_1 = gaussian_density_function(onetwo_mu_1, onetwo_variance_1, proj_x)
        gauss_2 = gaussian_density_function(onetwo_mu_2, onetwo_variance_2, proj_x)

        if (gauss_1/gauss_2) > (onetwo_prob_C_2/onetwo_prob_C_1):
            c1 = c1 + 1
        else:
            c2 = c2 + 1

        #13
        proj_x = np.matmul(onethree_w,x)
        gauss_1 = gaussian_density_function(onethree_mu_1, onethree_variance_1, proj_x)
        gauss_2 = gaussian_density_function(onethree_mu_3, onethree_variance_3, proj_x)

        if (gauss_1/gauss_2) > (onethree_prob_C_3/onethree_prob_C_1):
            c1 = c1 + 1
        else:
            c3 = c3 + 1

        #23
        proj_x = np.matmul(twothree_w,x)
        gauss_1 = gaussian_density_function(twothree_mu_2, twothree_variance_2, proj_x)
        gauss_2 = gaussian_density_function(twothree_mu_3, twothree_variance_3, proj_x)

        if (gauss_1/gauss_2) > (twothree_prob_C_3/twothree_prob_C_2):
            c2 = c2 + 1
        else:
            c3 = c3 + 1

        if max(c1,c2,c3) == c1:
            plt.scatter(x[0], x[1], c="red")
            #print("Class 1, x:", x)
        elif max(c1,c2,c3) == c2:
            plt.scatter(x[0], x[1], c="yellow")
            #print("Class 2, x:", x)
        elif max(c1,c2,c3) == c3:
            plt.scatter(x[0], x[1], c="green")
            #print("Class 3, x:", x)
        i = i + 1


    plt.plot()
    # for x in C_1:
    #     decision_function(w, mu_1, variance_1, mu_2, variance_2, prob_C_1, prob_C_2, x)

    # for x in C_2:
    #     decision_function(w, mu_1, variance_1, mu_2, variance_2, prob_C_1, prob_C_2, x)

    plt.scatter(dataset_1[:,0],dataset_1[:,1], c="red")
    plt.scatter(dataset_2[:,0],dataset_2[:,1], c="yellow")
    plt.scatter(dataset_3[:,0],dataset_3[:,1], c="green")
```

References

[Ng 2001] A. Y. Ng, M. I. Jordan, On Discriminative vs. Generative Classifiers: A Comparison of Logistic Regression and Naive Bayes, in: Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic, NIPS'01, MIT Press, Cambridge, MA, USA, 2001, pp. 841–848.

Task 3.2: PCA

We use the data-set iris.txt. Each row in it has 4 Attributes and a label(classification).

3.2a)

We normalize the data, s.t. every Attribute has mean zero and variance 1. Therefore, for every column, we compute the sample mean and subtract it. Then, we devide by the standard deviation. Of course, this is only done for the columns containing attributes.

```
#3a
n = len(iris)
pre_iris = iris[:,0:4]
pred = iris[:,4]
mean = pre_iris.mean(0)
step1 = pre_iris - mean
standard_deviation=np.std(pre_iris, axis=0)
normalized_data = np.multiply(step1, 1/standard_deviation).T
```

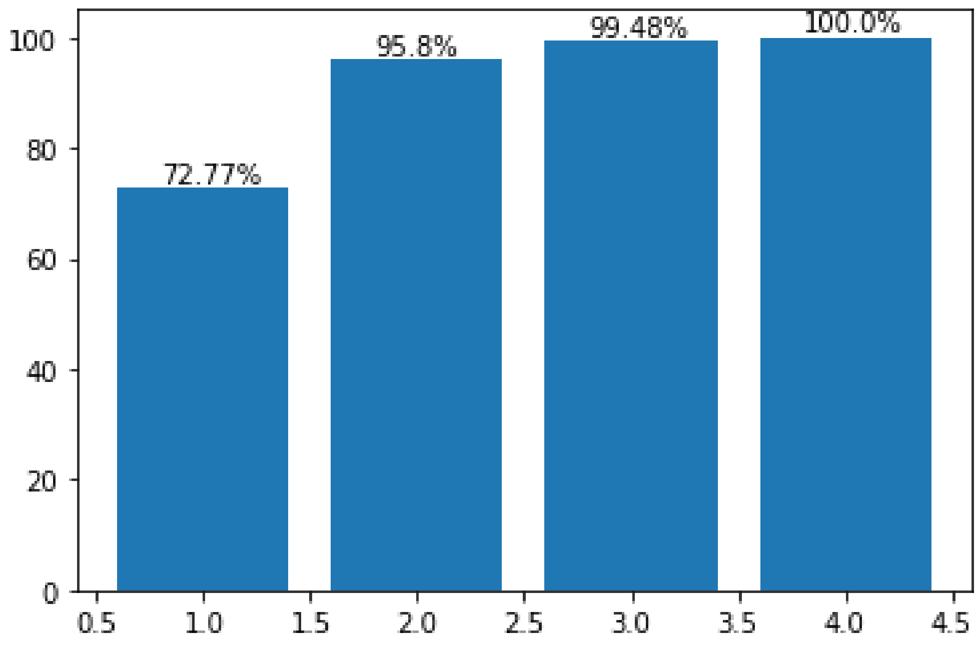
"normalized_data" does now contain one row per attribute, each row having mean zero and variance one.

3.2b)

We apply PCA on the normalized dataset. We plot a bar graph where the i-th bar shows how much of the original variance we already captured using the biggest i components.

```
def PCA(normalized_data):
    cov = np.cov(normalized_data)
    eigenvalues, eigenvectors = np.linalg.eig(cov)

    summe = np.sum(eigenvalues)
    eigenvalues_prop = eigenvalues/summe

    ind = [1,2,3,4]
    kum = np.zeros(4)
    kum[0] = eigenvalues_prop[0]
    for i in range(1,4):
        kum[i]=kum[i-1] + eigenvalues_prop[i]
    kum=kum*100
    plt.bar(ind, kum)
    rounded=np.round(kum,2)
    for idx,y in enumerate(rounded):
    plt.text(ind[idx]-0.2,y+1,str(y)+"%")
    plt.show()
    return eigenvalues, eigenvectors
```

Calling the function produces the following plot:



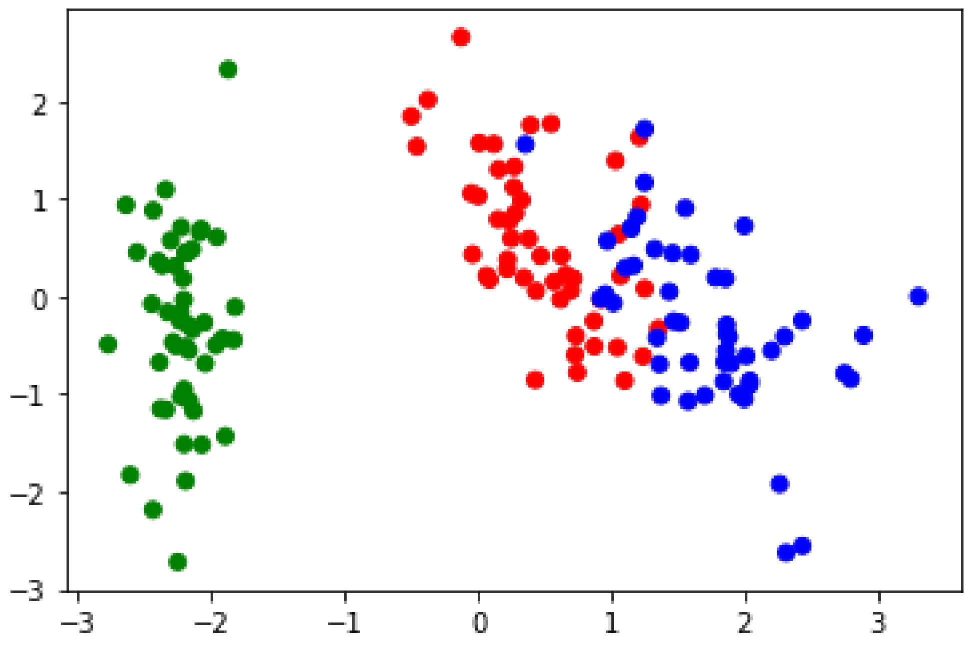
We see that for two components we explained already more than 95% of the variance.

3.2c)

We do the PCA for the two biggest components. We wrote the following function:

```
def threec(eigenvalues, eigenvectors, normalized_data, pred):
        B = eigenvectors[:,0:2]
        normalized_data_p = np.matmul(B.T, normalized_data)
        normalized_data_p = np.vstack((normalized_data_p, pred))
        colors = ['red','green','blue']
        plt.scatter(normalized_data_p[0], normalized_data_p[1],
                c=normalized_data_p[2], cmap=matplotlib.colors.ListedColormap(colors
        return
```

The first input arguments are clear, in the last argument we give the real data values from iris.txt just without the classification column. We then used the first two eigenvalues for projection on the R^2. Calling it, we get



where we used different colors for different classes(red=Setosa,green=Versicolour,blue=Virginica). We observe, that the green dots are well separated from the rest. Therefore we can argue, that Versicolour is pretty unique compared to the other two. For the red and blue dots it can be harsh to find a good decision boundary around x=1 we got both blue and red samples. The Setosa and the Virginica data seem to have more in common. Though, if we go away from the boundary, we could use the points to get - at least on the given training data - a very clear classification.

3.2d)

We perform the PCA for n=1,2,3,4 components. Then we take the computed points from R^n and embed them in R^4 to calculate an error distance. Doing this, we need to consider the normalization we have on our normalized_data, which was, for real dataset X with mean $\bar{x}$ and variance std^2

$$normalized\_data = \frac{X - \bar{x}}{std}$$

Following the idea of slide 27 in lecture 10, we get

$$\begin{aligned} a^n &= B^T \cdot (X - \bar{x}) \\ &= B^T \cdot (normalized\_data \cdot std) \end{aligned}$$

$$\tilde{x}^n = \bar{x} + B \cdot a^n$$

where B is the matrix of the first n eigenvalues. We implemented this backtransformation in the comp_set function.

```
def comp_set(n, eigenvectors, normalized_data, mean, var):
        B = eigenvectors[:,0:n+1]
        normalized_data_p = np.matmul(B.T, normalized_data)
        reconstruction = np.matmul(B, normalized_data_p)
        rec=np.multiply(reconstruction.T,var)
        rec=rec+mean
        return rec
```

Now, we can compare the backtransformed data to the original uncompressed data via normalized mean squared error.

```
def rmse(x):
        return np.sqrt(np.mean(x**2))

def threed(eigenvalues, eigenvectors, normalized_data, pre_iris, mean, var):
        lsg=np.zeros([4,4])
        for comp in range(4):
                re = comp_set(comp, eigenvectors, normalized_data, mean, var)
                diff = re - pre_iris
        for feature in range(4):
                lsg[comp, feature]=rmse(diff[:, feature])/
                        (np.sum(pre_iris[:, feature]/len(pre_iris)))
        print("lsg:", lsg)
        return
```

Calling the threed function gives us the following solution:

	x_1	x_2	x_3	x_4
n=1	6.406%	12.641%	6.021%	16.642%
n=2	3.945%	1.339%	5.946%	16.158%
n=3	0.531%	0.252%	5.381%	4.769%
n=4	0	0	0	0

3.2e)

1. Whitening is a technique to remove redundancies in our data. This is done by analyzing correlations between features and their variances. The covariance matrix Σ is very important in doing so, because if Σ is an identity matrix, we have no correlation between the features at all and every feature has normed variance of 1 which is the desired case for running stable algorithms on the data. In this case we call the dataset $\tilde{x}$ whitened. Because the resulting data is not unique, we can think of a somewhat 'best' transformation that still fits the requirements. The practical difference between PCA and ZCA lies in the interpretation of 'best' in the last sentence. Both are interested in normalizing the covariance as described, the PCA does this while also compressing the data. On the other hand, the ZCA is the better approach if we want to keep the new data pretty close to the original. We get it pretty easy from the PCA by multiplying an orthogonal matrix R from the left. The result still has the identity as covariance matrix. Choosing R as the matrix of eigenvectors from Σ will turn out to be optimal. We then call the matrix product $R \cdot \tilde{x}$ ZCA-whitened. Literature isn't clear about normalizing w.r.t. the means though. In 2. we give the idea of computation after normalization w.r.t. the mean, but e.g. brunner (see "https://cbrnr.github.io/2018/12/17/whitening-pca-zca/") does not normalize the means, but still gets effective results.

2. First of all we need to normalize every attribute of x, s.t. it has mean 0. We therefore estimate the mean by the sample mean $\hat{x}$:

$$x^* = x - \hat{x} = x - \frac{1}{n}\sum_{i=1}^{n} x_i$$

We compute the covariance matrix via

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}(x-\hat{x})^T(x-\hat{x}) = \frac{1}{n}\sum_{i=1} n x^{*T} x^*$$

and calculate an singular value decomposition $\Sigma = USV$, because we need the eigenvalues(now stored in S) and the eigenvectors (now stored in the columns of U). With this all done, we can compute the PCA whitened data:

$$x^{\text{PCAw}} = diag((diag(S) + \epsilon)^{-0.5}) \cdot U^T \cdot x*$$

where $diag$ extracts the diagonal of a matrix, or constructs a diagonal matrix given a vector. Finally, we can derive the ZCA via:

$$x^{\text{ZCAw}} = U \cdot x^{\text{PCAw}} \tag{4}$$

3. If we are given a new data example x, it will change the mean and therefore all parameters. In case we got big data we can assume that the single data point x has low impact on the mean and the computed covariance matrix. Then it is sufficient to use 4 for the single observation vector x and append the resulting column to x^{ZCAw}.

4. The following pyhton implementation follows the approach without normalization like presented in the webpage from brunner:

```python
def zca_brunner(x,epsilon):
        if x.shape[0]>x.shape[1]:
                raise
        evals,evecs=np.linalg.eigh(np.cov(x))
        evals=evals+epsilon
        z = evecs @ np.diag(evals**(-1/2)) @ evecs.T @ x
        return z
```

Alternatively, we could normalize with the mean and get

```python
def zca_whitening(epsilon):
        pre_iris = iris[:,0:4]
        mean = pre_iris.mean(0)
        xstern = (pre_iris - mean).T
        cov = np.cov(xstern)
        eigenvalues, eigenvectors = np.linalg.eigh(cov)
        xPCAwhite=np.diag(1./np.sqrt(eigenvalues+epsilon))@eigenvectors.T@xstern
        xZCAwhite=eigenvectors@xPCAwhite
        return xZCAwhite
```