Statistical Machine Learning: Exercise 3

Linear Regression, Linear Classification and Principal Component Analysis

Group 82: Alper Gece, Jinyao Chen

TECHNISCHE UNIVERSITÄT DARMSTADT

Summer Term 2021

Task 1: Linear Regression

1a) Linear Features

1. The ridge coefficients are used to be a reduced factor of the simple linear regression coefficients. It avoids overfitting, however, it adds a little deviation to the estimation.
2. A closed-form solution of the parameter estimation is used to derive the optimal model parameters by minimizing the squared error loss function. Closed form solution of the gradient descent algorithm is given below.

   $w = (\phi\phi^T + \lambda I)^{-1}\phi Y$

   With the help of the above function w values are found as below.
   $w(0) = 0.30946806$
   $w(1) = 1.19109478$
3. The codes that perform the above operations are given below.

```
import numpy as np
import math
import matplotlib.pyplot as plt

train_data = np.loadtxt("lin_reg_train.txt")
test_data = np.loadtxt("lin_reg_test.txt")

# get value of X and y
def Xy(data):
   n = len(data)
   X = np.zeros((n, 2))
   y = np.zeros((n, 1))
   for i in range(0, n):
      X[i][0] = data[i][0]
      X[i][1] = 1
      y[i][0] = data[i][1]

   return X, y

# lamda @ I
def lambda_I(c):
   I = np.zeros((2, 2))
   for i in range(0, 2):
      I[i][i] = c

   return I

# get w
```

```
def w(X, y, ci):
    return np.linalg.inv(X.T @ X + ci) @ X.T @ y

def predicted_value(x, w):
    y = np.empty((x.shape[0], 1))
    for i in range(0, x.shape[0]):
        y[i] = x[i] @ w

    return y

def RMSE(y_pre, y):
    n = len(y_pre)
    sum = 0
    for i in range(0, n):
        sum = sum + (y_pre[i] - y[i]) ** 2
    result = (sum / n) ** 0.5

    return result

def plot(x_real, y_real, y_predict):
    plt.scatter(x_real, y_real, c = '#A15949')
    plt.plot(x_real, y_predict, 'b-')
    plt.show()

if __name__ == '__main__':

    x, y = Xy(train_data)
    ci = lambda_I(0.01)
    w = w(x, y, ci)

    y_pre = predicted_value(x, w)
    rmse_train = RMSE(y_pre, y)
    print("rmse train is" + str(rmse_train))

    x_test, y_test = Xy(test_data)
    y_test_pre = predicted_value(x_test, w)
    rmse_test = RMSE(y_test_pre, y_test)
    print("rmse test is" + str(rmse_test))

    x_train_real = np.empty((len(x), 1))
    for i in range(len(x)):
        x_train_real[i] = x[i][0]
    plot(x_train_real, y, y_pre)
```

4. The below code reported the root mean squared error of the train and test data under our linear model with linear features.

   According to the code, the results are given below.

   The root mean squared error of train rmse is 0.41217802;
   The root mean squared error of test rmse is 0.38428817;

5. A plot that shows the training data as black dots and the predicted function as a blue line is given below.

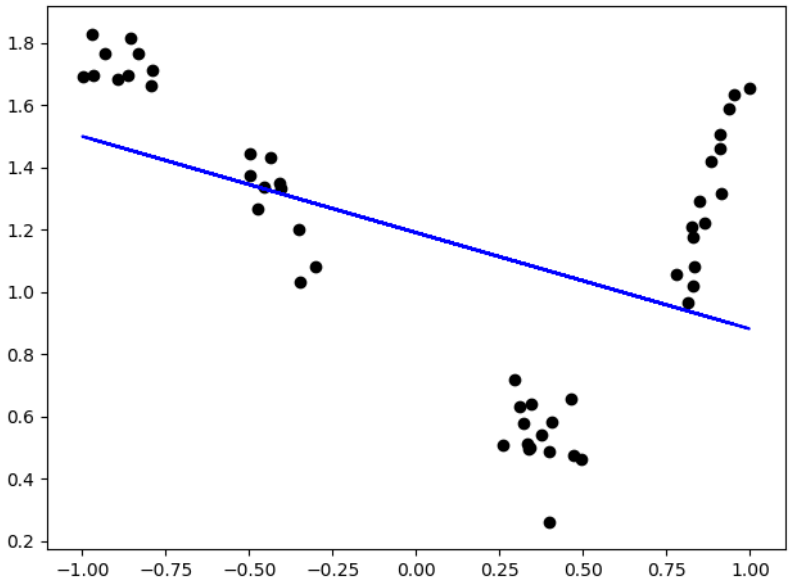


Figure 1: Linear Features

1b) Polynomial Features

1. The codes showing that how the polynomial features are generated and fitted to the model are given below.

```
import matplotlib.pyplot as plt
import numpy as np

train_data = np.loadtxt("lin_reg_train.txt")
test_data = np.loadtxt("lin_reg_test.txt")


def Xy(data, degree):
   n = len(data)
   d = degree + 1
   X = np.zeros((n, d))
   y = np.zeros((n, 1))
   for j in range(0, d):
      for i in range(0, n):
         if j == 0:
            X[i, j] = 1
         elif j == 1:
            X[i, j] = data[i, 0]
         else:
            X[i, j] = pow(data[i, 0], j)

   for i in range(0, n):
      y[i][0] = data[i][1]
   return X, y


def lambdaI(c, degree):
   d = degree + 1
   I = np.zeros((d, d))
   for j in range(0, d):
      I[j][j] = c
```

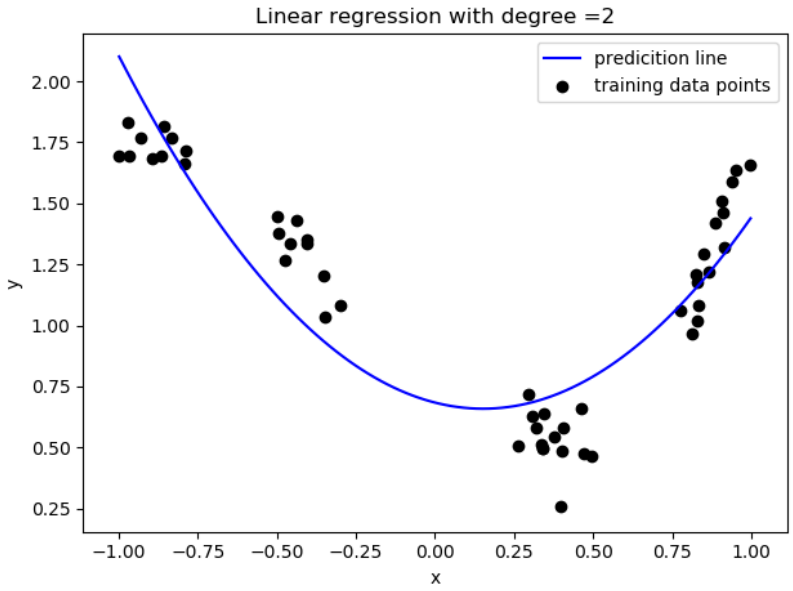
```
    return I


# get w
def w(X, y, ci):
    return np.linalg.inv(X.T @ X + ci) @ X.T @ y


def predicted_poly_value(x, w):
    y = np.empty((len(x), 1))
    for i in range(0, len(x)):
        y[i] = x[i] @ w

    return y


# calculate rmse
def RMSE_poly(y_pre, y):
    n = len(y_pre)
    sum = 0
    for i in range(0, n):
        sum = sum + (y_pre[i] - y[i]) ** 2
    r = (sum / n) ** 0.5

    return r


if __name__ == '__main__':

    x_train_real = np.zeros((len(train_data), 1))
    for i in range(len(train_data)):
        x_train_real[i] = train_data[i][0]

    for degree in range(2, 5):
        x, y = Xy(train_data, degree)
        x_test, y_test = Xy(test_data, degree)
        ci = lambdaI(0.01, degree)
        w_poly = w(x, y, ci)

        y_pre = predicted_poly_value(x, w_poly)
        y_pre_test = predicted_poly_value(x_test, w_poly)
        RMSE_poly_train = RMSE_poly(y_pre, y)
        RMSE_poly_test = RMSE_poly(y_pre_test, y_test)

        print("degree = " + str(degree))
        print("rmse train is" + str(RMSE_poly_train))
        print("rmse test is" + str(RMSE_poly_test))

        x_plot = np.linspace(np.min(x_train_real), np.max(x_train_real), num=100)
        x_plot = np.concatenate([x_plot.reshape(100, 1), np.ones(100).reshape(100, 1)],
            axis=1)
        x_plot_m, _ = Xy(x_plot, degree)
        # draw predicted curve
        y_plot = predicted_poly_value(x_plot_m, w_poly)
        # draw data dots
        plt.plot(x_plot.T[0], y_plot, c="blue", label='predicition line')
        plt.scatter(x_train_real, y, c="#A15949", label='training data points')
        plt.title("Linear regression with degree =" + str(degree))
        plt.xlabel("x")
        plt.ylabel("y")
```

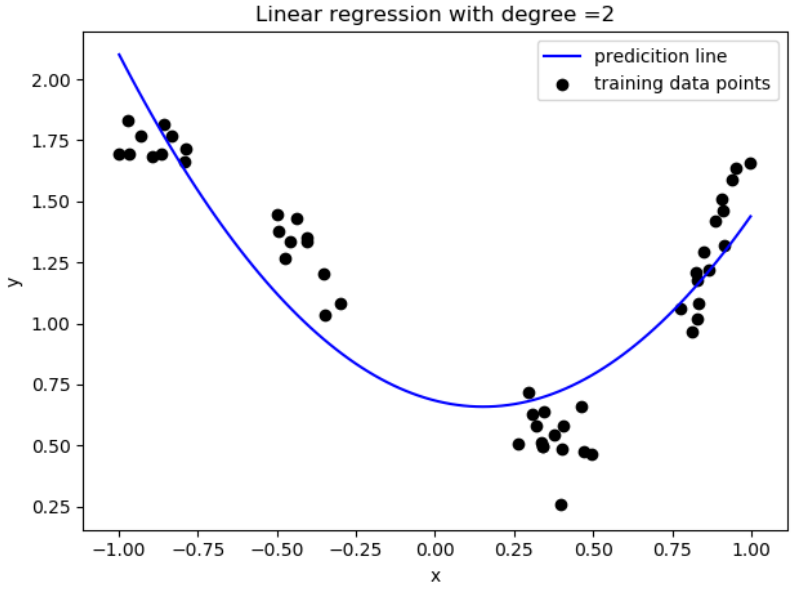
```
plt.legend()
plt.show()
```

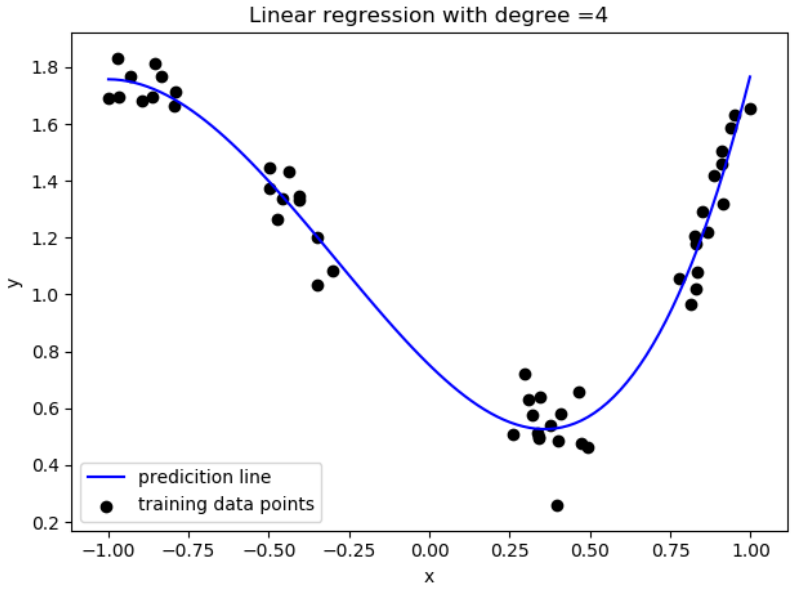
2. The below table shows the values that the root mean squared error of the training data and the testing data under the model with polynomial features.

$Degree$	$RMSETrainingData$	$RMSETestingData$
2	0.21201447	0.21687243
3	0.08706821	0.10835804
4	0.08701261	0.1066624

3. The below plots show the training data as black dots and the predicted function as a blue line.







Although polynomial regression fits a nonlinear model to the data, as a statistical estimation problem it is linear, in the sense that the regression function $E(y|x)$ is linear in the unknown parameters that are estimated from the data. For this reason, polynomial regression is considered to be a special case of multiple linear regression.

1c) Bayesian Linear Regression

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

   $P(w|X, y) \propto P(y|X, w)p(w)$

2. The predictive distribution $p(y_*|X_*, X, y)$ is stated below.

   $p(y_*|X_*, X, y) = \mathcal{N}(y_*|\mu(x_*), \sigma^2(x_*))$
   $\mu(x_*) = \phi^T(x_*)((\alpha/\beta)I + \phi\phi^T)^{-1}\phi y$
   $\sigma^2(x_*) = 1/\beta + \phi^T(x_*)(\alpha I + \beta\phi\phi^T)^{-1}\phi(x_*)$

3. The below code is showing how to compute the parameters of the posterior and predictive distribution.

```
import math
import matplotlib.pyplot as plt
import numpy as np

train_data = np.loadtxt("lin_reg_train.txt")
test_data = np.loadtxt("lin_reg_test.txt")

# get value of X and y
def Xy(data):
   n = len(data)
   X = np.zeros((2, n))
   y = np.zeros((n, 1))
   for i in range(0, n):
      X[0, i] = data[i, 0]
      X[1, i] = 1
      y[i, 0] = data[i, 1]

   return X, y
```

```
# lamda @ I
def lambda_I(alpha, beta):
    c = alpha / beta
    I = np.zeros((2, 2))
    for i in range(0, 2):
        I[i][i] = c

    return I

# get w
def parameter_posterior(X, y, ci):
    return np.linalg.inv(X @ X.T + ci) @ X @ y

def predicted_value(x, w):
    # y = np.empty((len(x), 1))
    # for i in range(0, len(y)):
    #    y[i] = x[i] @ w
    #
    # return y

    x_transpose = np.transpose(x)
    x_i=np.empty((1,2))
    y=np.empty((len(x_transpose),1))
    for i in range(0,len(y)):
        x_i=x_transpose[i]
        y[i]=np.matmul(x_i,w)
    return y

def RMSE(y_pre, y):
    n = len(y_pre)
    sum = 0
    for i in range(0, n):
        sum = sum + (y_pre[i] - y[i]) ** 2
    result = (sum / n) ** 0.5

    return result

def square(x_train, x_test, a, B):
    # lecture08 Page50
    # square = 1/B + x_test.T @ np.linalg.inv(B * x_train @ x_train.T + alphaI) @ x_test
    x_transpose = np.transpose(x_train)
    x_test_transpose = np.transpose(x_test)
    B_xx = B * (x_train @ x_transpose)
    square = np.zeros((len(x_test_transpose), 1))
    aI = np.zeros((2, 2))
    for j in range(0, 2):
        aI[j][j] = a
    inverse = np.linalg.inv((aI + B_xx))
    for i in range(0, len(square)):
        x = x_test_transpose[i]
        x_t = np.transpose(x)
        square[i] = (1/B) + x @ inverse @ x_t



    return square

def Gaussian(mean, square, y_data):
    p = np.empty((len(mean), 1))
    for i in range(0, len(square)):
```

```
        p1 = 1 / math.sqrt(2 * math.pi * square[i])
        p2 = ((-1) * pow((y_data[i] - mean[i]), 2)) / (2 * square[i])
        p[i] = p1 * math.exp(p2)

    return p

def average_log_likelihood(p):
    for i in range(len(p)):
        if i == 0:
            sum_y = np.log(p[i])
        else:
            sum_y = sum_y + np.log(p[i])

    average = sum_y / len(p)
    return average

if __name__ == '__main__':

    x_train, y_train = Xy(train_data)
    x_test, y_test = Xy(test_data)

    alpha = 0.01
    beta = 1 / (0.1 ** 2)
    ci = lambda_I(alpha, beta)
    w_posterior = parameter_posterior(x_train, y_train, ci)
    test_predicted_value = predicted_value(x_test, w_posterior)

    test_p = Gaussian(test_predicted_value, square(x_train, x_test, alpha, beta), y_test)
    log_l_test = average_log_likelihood(test_p)
    print("the log-likelihood of the test is"+str(log_l_test))
    print("rmse test is"+str(RMSE(test_predicted_value, y_test)))

    w_posterior_train = parameter_posterior(x_train, y_train, ci)
    train_predicted_value = predicted_value(x_train, w_posterior_train)
    train_p = Gaussian(train_predicted_value, square(x_train, x_train, alpha, beta),
        y_train)

    log_l_train = average_log_likelihood(train_p)
    print("the log-likelihood of the train is"+str(log_l_train))
    print("rmse train is" + str(RMSE(train_predicted_value, y_train)))

    x_ = np.linspace(np.min(x_train[0]), np.max(x_train[1]), num = 100).reshape(100, 1)
    x_ = np.concatenate([x_, np.ones(100).reshape(100, 1)], axis=1)
    y_ = predicted_value(x_.T, w_posterior)

    sig_ = square(x_.T, x_.T, alpha, beta)
    sig_ = np.sqrt(sig_)

    plt.scatter(x_.T[0], y_, c = 'blue', label = 'prediction')
    plt.scatter(x_train[0], y_train, c = 'black', label = 'original train data points')
    for i in range(3):
        plt.fill_between(x_.T[0], y_.reshape(100) + sig_.reshape(100) * (i+1),
                         y_.reshape(100) - sig_.reshape(100) * (i+1),
                         color = "b", alpha = 0.4)
    plt.title("Bayesian Linear Regression")
    plt.xlabel('x')
    plt.ylabel('y')

    plt.legend
    plt.show()
```

4. The RMSE of the train and test data are reported in the table below.

$RMSEtraindata$	$RMSEtestdata$
0.412	0.384

5. The average log-likelihood of the train and test data are reported in the table below.

$Log-likelihoodofthetraindata$	$Log-likelihoodofthetestdata$
0.412	0.384

6. The plot that 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 distribution in shades of blue is given below.

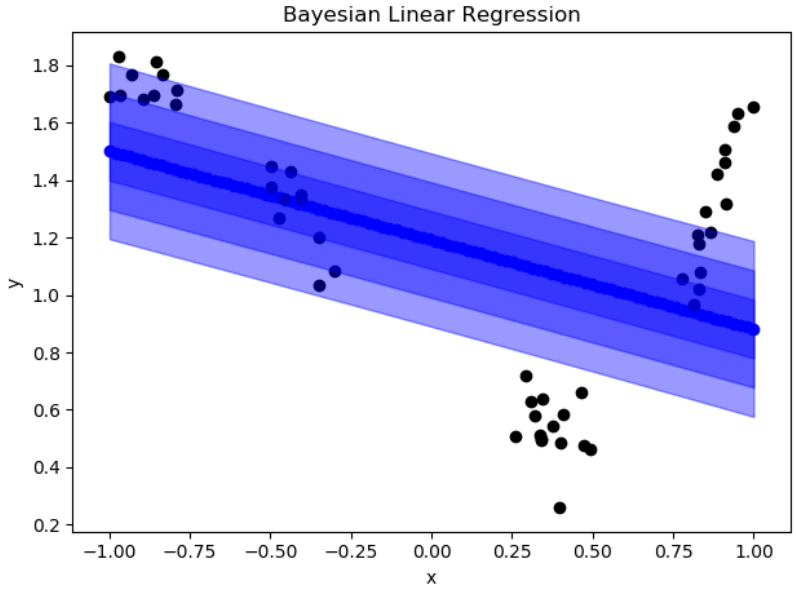


Figure 2: Bayesian Linear Regression

7. The weights obtained by linear regression and Bayesian linear regression are different since Bayesian linear regression uses the assumption of conditional independence. Therefore, Bayesian linear regression does not need to use gradient descent, but directly calculates the logical occurrence ratio of each feature as the weight. However, logical regression and conditional independence hypothesis are not valid. Coupling information between features can be obtained through gradient descent method, so as to obtain the corresponding weight.

1d) Squared Exponential Features

1. Codes that showing how to generate SE features and apply the Bayesian model are given below.

```
# phi_ij = e^(-0.5 * beta (X_i - alpha_j)^2)
def phiIJ(data):
   X_data = data[:, 0]
   y = data[:, 1]
   N = data.shape[0]
   k_dim = 20
   X_ = np.zeros([X_data.shape[0], 20])
   for i in range(X_data.shape[0]):
      for j in range(k_dim):
         X_[i][j] = np.power(np.e, ((-0.5 * B) * ((X_data[i] - (j + 1) * 0.1) ** 2)))

   b = np.ones(N)
```

```
    X_ = np.concatenate([X_, b.reshape(N, 1)], axis=1)
    return X_.T, y


def lambda_I(alpha, beta):
    c = alpha / beta
    I = np.zeros([21, 21])
    for j in range(0, 21):
        I[j, j] = c
    return I


# get w
def parameter_posterior(X, y, ci):
    return np.linalg.inv(X @ X.T + ci) @ X @ y


def predicted_value(x, w):
    y = np.empty((len(x.T), 1))
    for i in range(0, len(y)):
        x_i = x.T[i]
        y[i] = x_i @ w
    return y


def RMSE(y_pre, y):
    N = len(y_pre)
    sum = 0
    for i in range(0, N):
        sum = sum + pow((y_pre[i] - y[i]), 2)

    result = math.sqrt(sum / N)
    return result


def square(x_train, x_test, a, B):
    x_x = x_train @ x_train.T
    B_xx = B * x_x
    square = np.empty((len(x_test.T), 1))
    aI = np.zeros((B_xx.shape[0], B_xx.shape[0]))
    for j in range(0, B_xx.shape[0]):
        aI[j][j] = a
    inverse = np.linalg.inv((aI + B_xx))
    for i in range(0, len(square)):
        x = x_test.T[i]
        x_t = np.transpose(x)
        square[i] = (1 / B) + np.matmul((np.matmul(x, inverse)), x_t)

    return square


def Gaussian_Distribution(mean, square, y_data):
    p = np.empty((len(mean), 1))
    for i in range(0, len(square)):
        p1 = 1 / (math.sqrt(2 * math.pi * square[i]))
        p2 = ((-1) * pow((y_data[i] - mean[i]), 2)) / (2 * square[i])
        p[i] = p1 * math.exp(p2)
    return p
```

```
def average_log_likelihood(p):
    for i in range(len(p)):
        if i == 0:
            sumy = np.log(p[i])
        else:
            sumy = sumy + np.log(p[i])

    average = sumy / len(p)
    return average


if __name__ == '__main__':
    B = 1 / (0.1 ** 2)
    a = 0.01
    x_train_bayesian_ori, y_train_bayesian_ori = Xy(train_data)

    x_train, y_train = phiIJ(train_data)
    test_x, test_y = phiIJ(test_data)

    ci = lambda_I(a, B)
    w_posterior = parameter_posterior(x_train, y_train, ci)
    test_predicted_value = predicted_value(test_x, w_posterior)
    test_p = Gaussian_Distribution(test_predicted_value, square(x_train, test_x, a, B),
        test_y)
    log_l_test = average_log_likelihood(test_p)
    print("the log-likelihood of the test is" + str(log_l_test))
    print("RMSE test is" + str(RMSE(test_predicted_value, test_y)))

    w_posterior_train = parameter_posterior(x_train, y_train, ci)
    train_predicted_value = predicted_value(x_train, w_posterior_train)
    train_p = Gaussian_Distribution(train_predicted_value, square(x_train, x_train, a, B),
                            y_train)

    log_l_train = average_log_likelihood(train_p)
    print("the log-likelihood of the train is" + str(log_l_train))
    print("RMSE train is" + str(RMSE(train_predicted_value, y_train)))

    x_ = np.linspace(np.min(x_train_bayesian_ori[0]), np.max(x_train_bayesian_ori[0]),
        num=100).reshape(100, 1)
    x_ = np.concatenate([x_, np.ones(100).reshape(100, 1)], axis=1)
    x_maped, _ = phiIJ(x_)
    y_ = predicted_value(x_maped, w_posterior)

    sig_p = square(x_maped, x_maped, a, B)
    sig_p = np.sqrt(sig_p)

    plt.plot(x_.T[0], y_, c='blue', label='prediction')
    plt.scatter(x_train_bayesian_ori[0], y_train_bayesian_ori, c='black', label='original
        train data points')
    for i in range(3):
        plt.fill_between(x_.T[0], y_.reshape(100) + sig_p.reshape(100) * (i + 1.),
                         y_.reshape(100) - sig_p.reshape(100) * (i + 1.),
                         color="b", alpha=0.2)
    plt.title("Bayesian Linear Regression ")
    plt.xlabel('x')
    plt.ylabel('y')

    plt.legend()
    plt.show()
```

2. The RMSE of the train and test data are reported in the table below.

$RMSEtraindata$	$RMSEtestdata$
0.167	0.962

3. The average log-likelihood of the train and test data are reported in the table below.

$Log-likelihoodofthetraindata$	$Log-likelihoodofthetestdata$
-0.003	-0.9886

4. The plot that 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 distribution in shades of blue is given below.

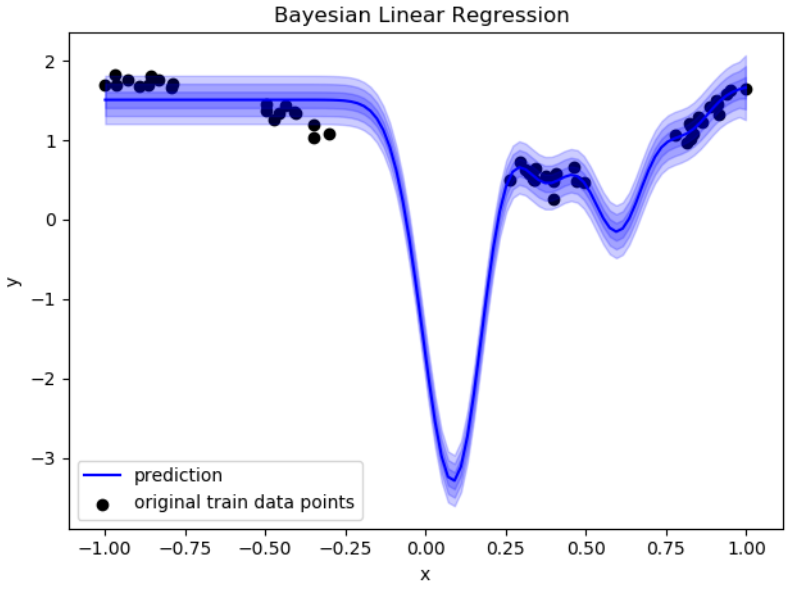


Figure 3: Squared Exponential Features

5. SE features is similar to gaussian distribution. α is like mean, β is reciprocal of variance.

1e) Cross validation

Cross validation is any of various similar model validation techniques for assessing how the results of a statistical analysis will generalize to an independent data set.

Pros:

- It validates the performance of your model on multiple folds of your data.
- It can balance out the predicted features classes if you are dealing with an unbalanced dataset.
- Because of 1 and 3, it gives you a more stable answer as to how your model performs on that data.

Cons:

- K-fold does not really work well with sequential data (a time series of some kind). If you use it you would be predicting past values with future value training, which is not a good way to go about things.
- If you just trust the K-fold final aggregated score for your model, imagine an r-squared of 0.95, you will miss a lot of information about your models' performance like the spikes I described above.

Task 2: Linear Classification

2a) Discriminative and Generative Models

The generative model is used for modeling the class-conditional distributions $p(x|C2)$ and $p(x|C1)$. It classifies the class posterior using Bayes' rule. One example can be "Naive Bayes" to generative model.

The discriminative model is used for modeling the class-posterior directly, such as $p(C1|x)$. This model concerns getting the classification right instead of fitting the class-conditional well. One example can be "Logistic Regression" to discriminative model.

Learning in a discriminative model is easier since there is no search for the probability distribution. The discriminative model doesn't need much computation because with it we can just classify something. Compared to the discriminative model, generating model has higher generalization ability and universality, which means higher computational complexity, and it can help to discover new features in the data.

2b) Linear Discriminant Analysis

In this subtask, Linear Discriminant Analysis is used to classify the points in the dataset. Related codes and figures are given below. Number of misclassified samples are 28.

```
def lda(data):
   x1 = data[:50].values
   x2 = data[50:(50 + 44)].values
   x3 = data[(50 + 44):].values
   y = [0] * 50 + [1] * 44 + [2] * 43
   c = []
   for label in y:
      if label == 0:
         c.append('yellow')
      elif label == 1:
         c.append('blue')
      elif label == 2:
         c.append('red')

   plt.figure()
   plt.title("Data points with original classes")
   plt.xlabel("x1")
   plt.ylabel("x2")
   plt.scatter(data.values[:, 0], data.values[:, 1], c=c)
   c1 = mpatches.Patch(color='yellow', label='Class 1')
   c2 = mpatches.Patch(color='blue', label='Class 2')
   c3 = mpatches.Patch(color='red', label='Class 3')
   plt.legend(handles=[c1, c2, c3], loc='lower right')
   plt.show()

   x = np.array([x1, x2, x3])
   n_samples = np.array([x1.shape[0], x2.shape[0], x3.shape[0]])

   means = np.concatenate((np.mean(x1, axis=0),
                           np.mean(x2, axis=0),
                           np.mean(x3, axis=0))
                          ).reshape(x.shape[0], data.values.shape[1])

   # class_mean = np.sum(n_samples[:, np.newaxis] * means, axis=0) / data.shape[0]
   total_mean = np.mean(data.values, axis=0)
   S_total = np.zeros(shape=(means.shape[1], means.shape[1]))
```

```
S_within = np.zeros(shape=(means.shape[1], means.shape[1]))
S_between = np.zeros(shape=(means.shape[1], means.shape[1]))

# total scatter
for val in data.values:
   diff = val - total_mean
   S_total += np.outer(diff, diff.T)

# within class scatter
for i, subset in enumerate(x):
   S_k = np.zeros(shape=(means.shape[1], means.shape[1]))
   for val in subset:
      diff = val - means[i]
      S_k += np.outer(diff, diff.T)

   # S_k = np.cov(subset, rowvar=False)
   S_within += S_k

S_total /= data.values.shape[0]
S_within /= data.values.shape[0]

# between class scatter
S_between = S_total - S_within # Bishop page 191, equation (4.45)

# eigendecomposition
eigenval, eigenvec = np.linalg.eig(np.linalg.inv(S_within).dot(S_between))

# select first c-1 eigenvectors
idx = (-eigenval).argsort()
W = eigenvec[idx][:, :means.shape[1] - 1].T

# calculate projections and their mean / variance
proj = []
mean_proj = np.empty(shape=(means.shape[0], means.shape[1] - 1))
# covar_proj = np.empty(shape=(means.shape[0], x1.shape[1], x1.shape[1]))
var_proj = np.empty(shape=(means.shape[0], means.shape[1] - 1))

for i, subset in enumerate(x):
   proj.append(W.dot(subset.T))
   mean_proj[i] = np.mean(proj[i], axis=1)
   var_proj[i] = np.sum((proj[i] - mean_proj[i]) ** 2) / (subset.shape[0] - 1)

prior = n_samples / np.sum(n_samples)

# gaussian maximum likelihood posterior
proj_flat = np.concatenate([proj[0][0], proj[1][0], proj[2][0]])
gaussian_posterior = np.empty(shape=(len(x), data.shape[0]))
for i in range(mean_proj.shape[0]):
   gaussian_posterior[i] = np.log(gaussian(proj_flat, mean_proj[i], var_proj[i]) *
       prior[i])

y_hat = gaussian_posterior.argmax(axis=0)
plt.figure()
plt.title("Data points with LDA classification")
plt.xlabel("x1")
plt.ylabel("x2")
c = []
for label in y_hat:
   if label == 0:
      c.append('yellow')
```

```
        elif label == 1:
            c.append('blue')
        elif label == 2:
            c.append('red')

    plt.legend(handles=[c1, c2, c3], loc='lower right')
    plt.scatter(data.values[:, 0], data.values[:, 1], c=c)
    plt.show()

    print('Number of misclassified samples:', np.count_nonzero(y - y_hat))


def gaussian(data, mu, var):
    out = np.empty(data.shape[0])
    denom = np.sqrt(2 * np.pi * var)
    for i, val in enumerate(data):
        out[i] = np.exp(-(val - mu) ** 2 / (2 * var)) / denom

    return out


def multivariate_gaussian(data, mu, covar):
    out = np.empty(data.shape[0])
    denom = np.sqrt((2 * math.pi) ** data.shape[1] * np.linalg.det(covar))

    # compute for each data point
    for i, x in enumerate(data):
        diff = x - mu
        out[i] = np.exp(-.5 * diff.dot(np.linalg.inv(covar)).dot(diff.T)) / denom

    return out
```

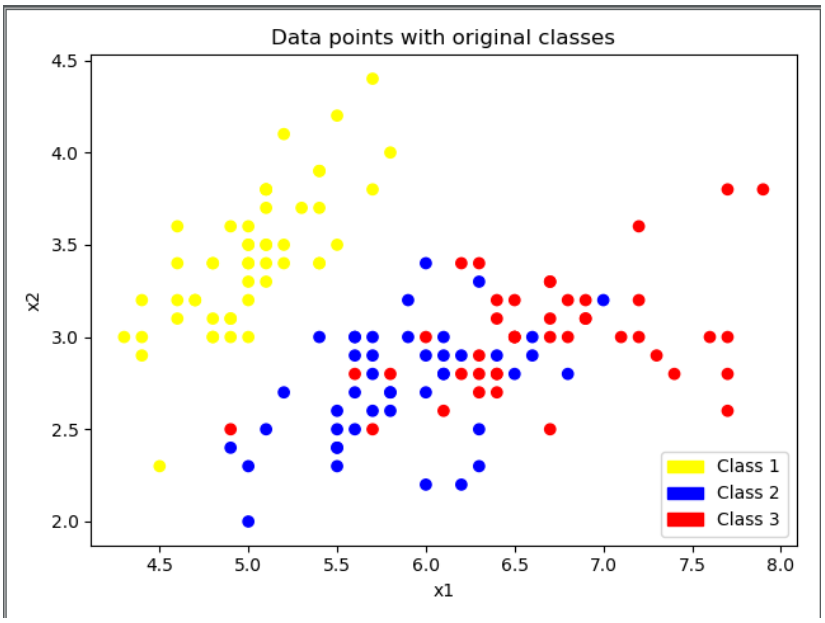


Figure 4: Data Points with original classes

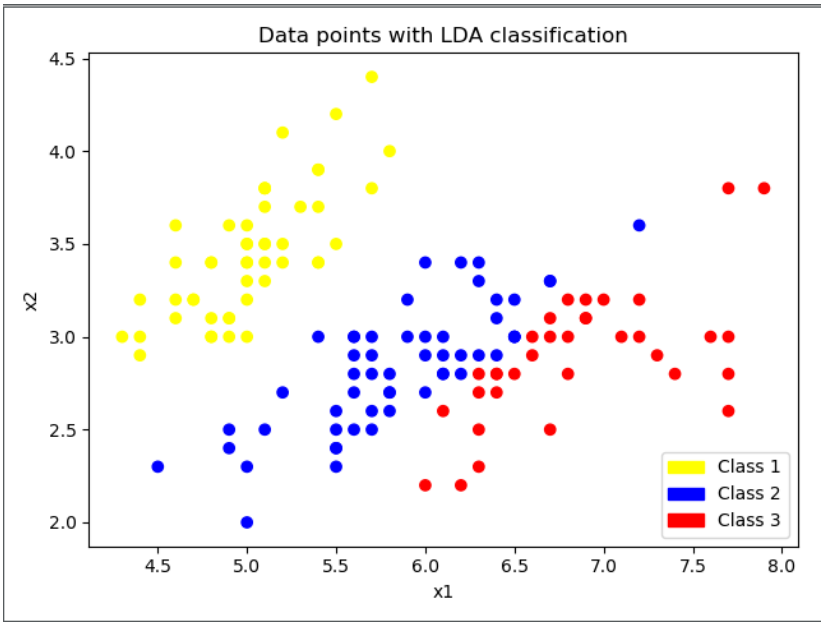


Figure 5: Data Points with LDA classificatio

Task 3: Principal Component Analysis

3a) Data Normalization

Mainly for the convenience of data processing put forward, the data mapping to a certain range of processing, more convenient and fast. Change a dimension expression to a dimensionless expression so that indicators of different units or orders of magnitude can be compared and weighted. Normalization is a way of simplifying the calculation. In essence, normalization/standardization is a linear transformation. Linear transformation has many good properties, which determine that the change of data will not cause "failure", but can improve the performance of data. These properties are the premise of normalization/standardization. One important property, for example, is that a linear transformation does not change the numerical order of the original data. Because of these reasons, normalizing the provided dataset is so significant for the machine learning algorithms. The sample code of normalizing is given below.

```
def normalize(x):
   return (x - np.mean(x, axis=0)) / np.std(x, axis=0)
```

3b) Principal Component Analysis

PCA is applied to the normalized dataset. Related codes and plot which is showing the proportion of the cumulative variance explained are given below.

```
def PCA(x, n_eigen):
   C = x.dot(x.T) / x.shape[1]
   eigenvalues, eigenvectors = np.linalg.eig(C)
   eigenvalues_total = np.sum(eigenvalues)

   # eigenvalues are already sorted
   explained = np.sum(eigenvalues[:n_eigen + 1]) / eigenvalues_total

   B = eigenvectors[:, :n_eigen + 1]
```

```
    a = B.T.dot(x)

    return B, a, explained

def find_threshold(x):
    max_eigenvalue = x.shape[1]

    explained = np.empty(shape=(max_eigenvalue,))
    for n_eigen in range(max_eigenvalue):
        _, a, var = PCA(normalize(x), n_eigen)
        explained[n_eigen] = var

    plt.plot(np.arange(1, max_eigenvalue + 1), explained, label="marginal variance captured")
    plt.plot(np.arange(1, max_eigenvalue + 1), np.full(max_eigenvalue, 0.95), "--",
        label="threshold $\lambda=0.95$")
    plt.xticks(np.arange(1, max_eigenvalue + 1))
    plt.title("Threshold of marginal variance captured")
    plt.xlabel("# eigenvectors")
    plt.ylabel("marginal variance captured")
    plt.legend()
    plt.show()
```

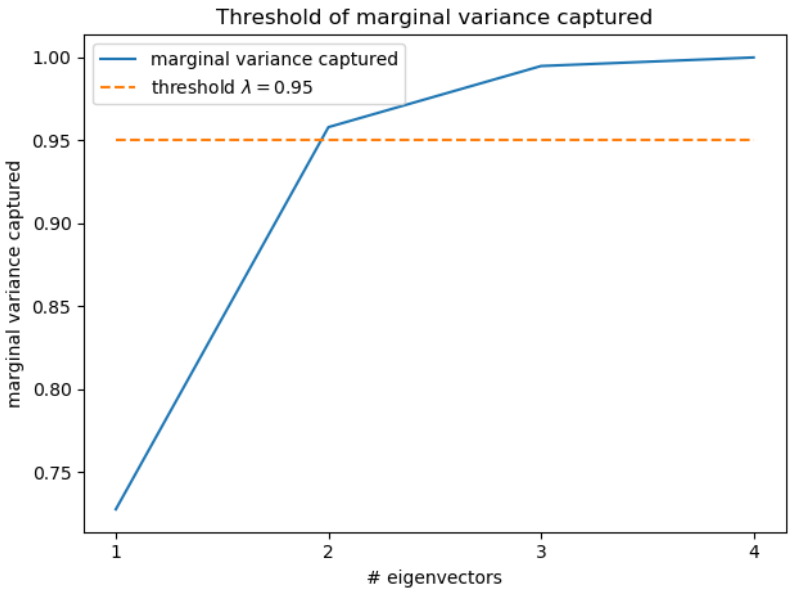


Figure 6: Threshhold of marginal variance captured

At least 2 eigenvectors are needed in order to explain at least 95% of the dataset variance.

3c) Low Dimensional Space

Two components are used to explain 95% of the dataset variance. A separation between the classes purple and yellow can not be done precisely since coinciding exists. However, the separation between green and other classes is easier. Codes and plot of the lower-dimensional projection of the data are given below.

```
def plot_PCA_data(x, y):
    max_eigenvalue = x.shape[1]

    projection = None
```

```
for n_eigen in range(max_eigenvalue):
    _, a, var = PCA(normalize(x), n_eigen)
    if var < .95: continue
    projection = a
    break

# colors = [int(i % np.unique(y).shape[0]) for i in y]
plt.scatter(projection[0], projection[1], c=y)
plt.title("Data points after PCA")
plt.xlabel("x1")
plt.ylabel("x2")
plt.show()
```

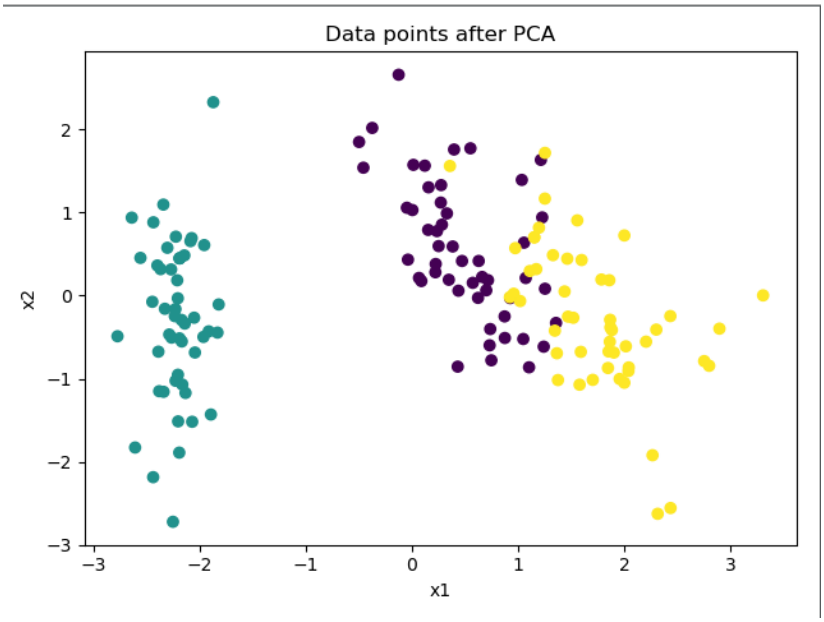


Figure 7: Data Points after PCA

3d) Projection to the Original Space

The original dataset is reconstructed and the normalized root mean square error (NRMSE) is used as a metric to fill the table. Codes and table are given below.

```
def reverse_PCA(x):
   mean = np.mean(x, axis=0)
   std = np.std(x, axis=0)

   max_eigenvalue = x.shape[1]

   nrmse_total = np.empty(shape=(max_eigenvalue, x.shape[1]))

   for n_eigen in range(max_eigenvalue):
      B, a, _ = PCA(normalize(x), n_eigen)
      norm = np.amax(x, axis=0) - np.amin(x, axis=0)
      reverse = std * B.dot(a).T + mean
      nrmse_total[n_eigen] = nrmse(reverse, x, norm)

   print(nrmse_total)
```

$Nofcomponents$	x_1	x_2	x_3	x_4
1	$1.03979363e-01$	$1.60861958e-01$	$3.83578329e-02$	$8.31176493e-02$
2	$6.40336901e-02$	$1.70340973e-02$	$3.78801525e-02$	$8.07006820e-02$
3	$8.62221480e-03$	$3.20869369e-03$	$3.42790326e-02$	$2.38189270e-02$
4	$9.44851217e-17$	$1.07894284e-16$	$1.78278775e-16$	$1.07902548e-16$

The table shows that when the more components are used, the less information is lost.

3e) Kernel PCA

Kernel PCA is an improved version of PCA, which converts non-linear separable data into a new low-dimensional subspace suitable for alignment and linear classification by using techniques of kernel methods. Kernel PCA can transform data into a high-dimensional space through nonlinear mapping, and then uses PCA to map it to another low-dimensional space in the high dimensional space. The samples are divided by a linear classifier. Kernel PCA uses a kernel function to project data-set into a higher dimensional feature space, where it is linearly separable.

Kernel PCA generally has higher memory and runtime requirements than PCA, and can hardly be scaled to massive datasets so that it brings high computational cost. Various strategies exist for scaling up Kernel PCA, but this requires making approximations.

References

[1] Zak Jost: bayesian-linear-regression,
`https://github.com/zjost/bayesian-linear-regression/blob/master/src/bayes-regression.ipynb`

[2] Wikipedia: Cross Validation,
`https://en.wikipedia.org/wiki/Cross-validation_(statistics)`

[3] Emil Filipov: Advantages and Disadvantages of k fold cross validation,
`https://www.quora.com/What-are-the-advantages-and-disadvantages-of-k-fold-cross-validation`