Statistical Machine Learning: Exercise 3

Group A: Kexin Wang (2540047), Paul Philipp Seitz (2337506)

TECHNISCHE UNIVERSITÄT DARMSTADT

Summer Term 2022

Task 1: Linear Regression(44Points)

1a) Linear Features (12 Points)

1).
The ridge coefficient is a reduced factor of the simple linear regression coefficients that never attains zero values but very small values. It is used to prevent overfitting of the data and improve the numerical stability.

2).[1]
Due to the quadratic term in the squared error function, a larger difference between the estimated value and the observed value will contribute a larger loss. First, when training the model, we want to find the parameter $\hat{\mathbf{w}}$, which minimizes the total loss over all training samples. We need to minimize the squared Euclidean norm of $\mathbf{w}$, scaled by the ridge coefficient λ.

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \| \mathbf{Xw\text{-}y} \|_2^2 + \lambda \| \mathbf{w} \|_2^2$$

Second, setting the derivative of the loss with respect to $\mathbf{w}$ to 0 yields the analytical solution.

$$\frac{\partial}{\partial \mathbf{w}} \| \mathbf{Xw\text{-}y} \|_2^2 + \lambda \| \mathbf{w} \|_2^2 = 0$$
$$\Rightarrow 2\mathbf{X}^\top\mathbf{XW} - 2\mathbf{X}^\top\mathbf{Y} + 2\lambda\mathbf{w} = 0$$

Finally, solving the equation for $\mathbf{w}$ yields.

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

3).

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

test = np.loadtxt('dataSets/lin_reg_test.txt')
train = np.loadtxt('dataSets/lin_reg_train.txt')

x_train = train[:, :-1]
y_train = train[:, -1]
x_test = test[:, :-1]
y_test = test[:, -1]

def rmse(y_pred, y):
    return np.sqrt(np.mean((y_pred - y)**2))

def linear_ridge_regression(plot_path=None, ridge = 0.01):
    # See 1a.2
    n = x_train.shape[1]
```

```
    w = np.linalg.inv(x_train.T @ x_train + ridge * np.eye(n)) @ x_train.T @ y_train

    y_pred_train = x_train @ w
    y_pred_test = x_test @ w

    train_rmse = rmse(y_pred_train, y_train)
    test_rmse = rmse(y_pred_test, y_test)

    print('Train RMSE: ', train_rmse)
    print('Test RMSE: ', test_rmse)

    x = np.arange(-1, 1, 1e-2)
    y_pred = x * w[0] + w[1]

    plt.plot(train[:, :-1], y_train, 'o', color='k', label='Training data')
    plt.plot(x, y_pred, color='tab:blue', label='Linear prediction')
    plt.legend()
    plt.xlabel('x')
    plt.ylabel('y')
    if plot_path:
        plt.savefig(plot_path, bbox_inches='tight')
    else:
        plt.show()


# Append one dimension for the bias term
x_train = np.append(x_train, np.ones((x_train.shape[0], 1)), axis=1)
x_test = np.append(x_test, np.ones((x_test.shape[0], 1)), axis=1)
linear_ridge_regression('1a.png')
```

4).

```
Train RMSE: 0.4121780156736108
Test RMSE:  0.38428816992597875
```

5).

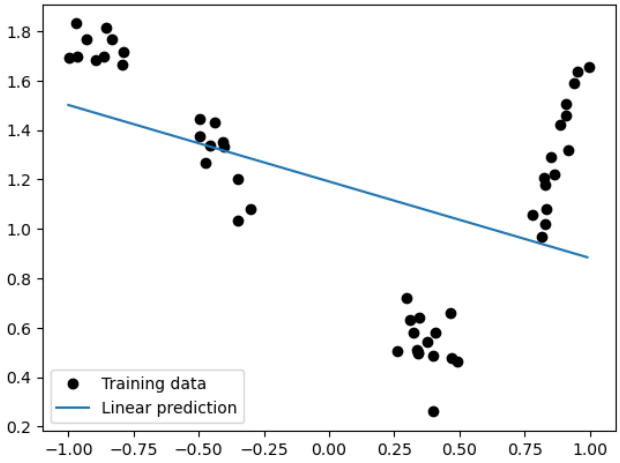

Figure 1: Training data and linear prediction

1b) Polynomial Features (8 Points)

1).

```python
def linear_ridge_regression(phi, d, plot_path=None, ridge = 0.01):
    # See 1a.2
    phi_train = phi(x_train)
    phi_test = phi(x_test)

    n = phi_train.shape[1]
    w = np.linalg.inv(phi_train.T @ phi_train + ridge * np.eye(n)) @ phi_train.T @ y_train

    y_pred_train = phi_train @ w
    y_pred_test = phi_test @ w

    train_rmse = rmse(y_pred_train, y_train)
    test_rmse = rmse(y_pred_test, y_test)

    print('Train RMSE (d={}): '.format(d), train_rmse)
    print('Test RMSE (d={}): '.format(d), test_rmse)

    x = np.arange(-1, 1, 1e-2)
    y_pred = phi(x) @ w

    plt.plot(x_train, y_train, 'o', color='k', label='Training data (d={})'.format(d))
    plt.plot(x, y_pred, color='tab:blue', label='Linear prediction (d={})'.format(d))
    plt.legend()
    plt.xlabel('x')
    plt.ylabel('y')

    if plot_path:
        plt.savefig(plot_path, bbox_inches='tight')
        plt.clf()
    else:
        plt.show()

def polynomial_feature_projection(x, degree):
    # Polynomial projection: x -> [1, x, x^2, ..., x^degree]
    poly = np.array([np.power(x, i) for i in range(degree+1)]).T
    return poly.reshape([x.shape[0], degree+1])


for d in [2,3,4]:
    poly_phi = lambda x: polynomial_feature_projection(x, d)
    linear_ridge_regression(poly_phi, d, '1b_d{}.png'.format(d))
```

2).

```
Train RMSE (d=2):   0.21201447265968612
Test RMSE (d=2):    0.21687242714148727
Train RMSE (d=3):   0.08706821295481752
Test RMSE (d=3):    0.10835803719738049
Train RMSE (d=4):   0.08701261306638186
Test RMSE (d=4):    0.10666239820964747
```

3).

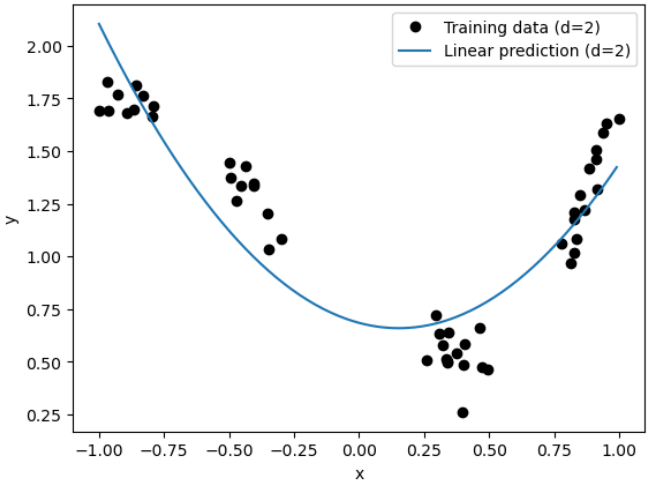

Figure 2: Training data and polynomial prediction (d=2)

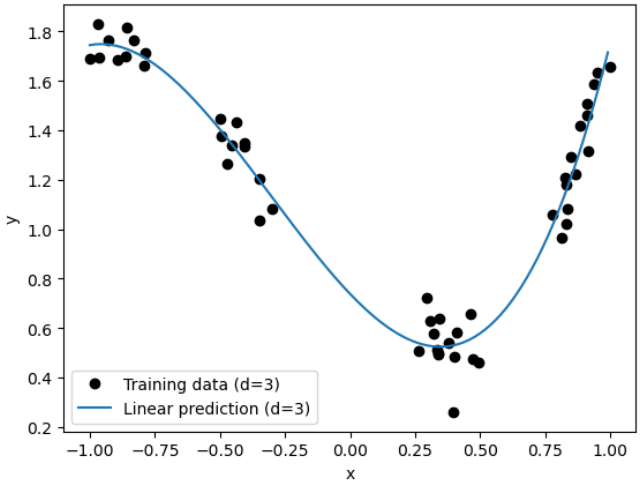

Figure 3: Training data and polynomial prediction (d=3)

4).
Even though the features are projected to a polynomial space as a pre-processing step, the regression itself is still a linear function as it is composed of linear operations (multiplication with weight matrix and addition of bias).

1c) Bayesian Linear Regression (11 Points)

1).

$$p(\mathbf{w}|\mathbf{X},\mathbf{y}) = \mathcal{N}(\mu_n, \Lambda_n^{-1}).$$

$$\sigma = 0.1, \lambda = 0.01$$

$$\mu_n = \sigma^{-2} * \Lambda_n^{-1}\mathbf{X}^T\mathbf{y}$$

$$\Lambda_n^{-1} = \sigma^{-2}\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}$$

2).

$$p(\mathbf{y}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{y}) = \int p(\mathbf{y}_*|\mathbf{X}_*, \mathbf{w})p(\mathbf{w}|\mathbf{X}, \mathbf{y})\mathrm{d}w = \mathcal{N}(\mathbf{X}_*\mu_n, \sigma^2 + \mathbf{X}_*\Lambda_n^{-1}\mathbf{X}_*^T)$$

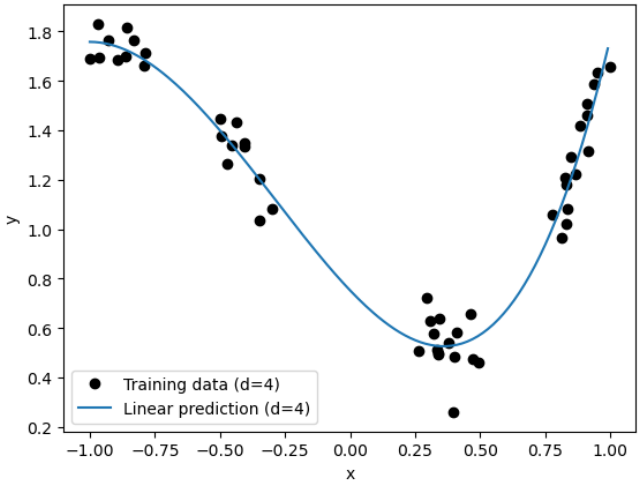

Figure 4: Training data and polynomial prediction (d=4)

3).

```
def extend_x(data):
    # Append one dimension for the bias term
    n = data.shape[0]
    X = np.vstack([data[:, 0], np.ones((1, n))])
    y = data[:, 1]

    return X, y

def compute_w(X, y, ridge):
    # Compute w from posterior distribution (see 1c.1)
    return np.linalg.inv(X @ X.T + ridge) @ X @ y

def predict_y(x, w):
    # Compute y_pred from predictive distribution (see 1c.2)
    return np.array([x @ w for x in x.T])

def rmse(y_pred, y):
    return np.sqrt(np.mean((y_pred - y)**2))

def compute_var(x_train, x_test, a, beta):
    inverse = np.linalg.inv((a * np.eye(x_train.shape[0]) + beta * (x_train @ x_train.T)))
    sigma = [(1/beta) + x @ inverse @ x.T for x in x_test.T]
    return sigma

def gaussian_distance(y, y_pred, sigma):
    # Compute gaussian distance for each (y, y_pred, sig)
    p = [1 / np.sqrt(2 * np.pi * sig) * np.exp(((-1) * pow((y - m), 2)) / (2 * sig)) for y, m,
        sig in zip(y, y_pred, sigma)]
    return p


train = np.loadtxt('dataSets/lin_reg_train.txt')
test = np.loadtxt('dataSets/lin_reg_test.txt')
x_train, y_train = extend_x(train)
x_test, y_test = extend_x(test)

# Compute ridge
d = x_train.shape[0]
```

```
a = 0.01
std = 0.1
beta = (std**2)**(-1)
ridge = a / beta * np.eye(d)

w = compute_w(x_train, y_train, ridge)
train_y_pred = predict_y(x_train, w)
test_y_pred = predict_y(x_test, w)

# 1c.4
print("Train RSME: ", str(rmse(train_y_pred, y_train)))
print("Test RSME: ", str(rmse(test_y_pred, y_test)))

# 1c.5
var = compute_var(x_train, x_test, a, beta)
train_log_likelihood = np.mean(np.log(gaussian_distance(y_train, train_y_pred, var)))
test_log_likelihood = np.mean(np.log(gaussian_distance(y_test, test_y_pred, var)))

print("Train Log Likelihood: ", str(train_log_likelihood))
print("Test Log Likelihood: ", str(test_log_likelihood))

# 1c.6
x = np.linspace(-1, 1, 1000)
x = np.vstack([x, np.ones(1000)])

y_pred = predict_y(x, w)

plt.scatter(x_train[0], y_train, c='k', label = 'Training data')
plt.plot(x[0], y_pred, c='tab:blue', label = 'Function prediction')

# Plot 1, 2 and 3 standard deviations in different shades
plt.fill_between(x[0], y_pred - std, y_pred + std, color='tab:blue', alpha=1/3)
plt.fill_between(x[0], y_pred - 2 * std, y_pred + 2 * std, color='tab:blue', alpha=1/3)
plt.fill_between(x[0], y_pred - 3 * std, y_pred + 3 * std, color='tab:blue', alpha=1/3)

plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.savefig('1c.png', bbox_inches='tight')
plt.clf()
```

4).

```
Train RSME:  0.4121779259165973
Test RSME:  0.38434085452132943
```

5).

```
Train Log Likelihood:  -6.867540282525448
Test Log Likelihood:  -5.774752732594602
```

6).

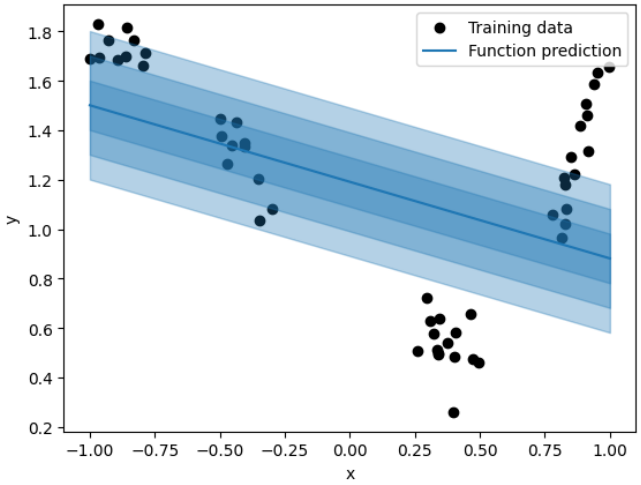

Figure 5: Training data and bayesian prediction

7).

Linear regression:

The optimal value of the parameter $\hat{\mathbf{w}}$ is calculated by setting the gradient of the squared error loss function to zero, which means that the linear regression only calculates a single vector for $\mathbf{w}$.

Bayesian linear regression:

We define the data likelihood $p(\mathcal{D}|\mathbf{w})$, the prior $p(\mathbf{w})$ and the marginal likelihood $p(\mathcal{D}) = \int p(\mathcal{D}|\mathbf{w})p(\mathbf{w})\mathrm{d}w$. By computing these values, we can compute the full posterior distribution of parameters $\mathbf{w}$. Thus the Bayesian linear regression computes a full probability distribution for $\mathbf{w}$.

1d) Squared Exponential Features (9 Points)

1).

```
def squared_exponential(data, k, beta):
    n = data.shape[0]
    X = data[:, 0]
    y = data[:, 1]

    alphas = [j * 0.1 - 1 for j in range(k)]
    X_se = np.array([np.exp(-beta/2 * (Xi - alphaj)**2) for Xi in X for alphaj in alphas])

    # Append one dimension for bias
    X_se = np.vstack([X_se.reshape((n,k)).T, np.ones(n)])
    return X_se, y


train = np.loadtxt('dataSets/lin_reg_train.txt')
test = np.loadtxt('dataSets/lin_reg_test.txt')

a = 0.01
beta = 10
std = 0.1
k = 20
```

```
ridge = a / beta * np.eye(k + 1)

x_train, y_train = squared_exponential(train, k, beta)
x_test, y_test = squared_exponential(test, k, beta)

# 1d.3
w = compute_w(x_train, y_train, ridge)
train_y_pred = predict_y(x_train, w)
test_y_pred = predict_y(x_test, w)

# 1d.4
print("Train RSME: ", str(rmse(train_y_pred, y_train)))
print("Test RSME: ", str(rmse(test_y_pred, y_test)))

# 1d.5
var = compute_var(x_train, x_test, a, beta)
train_log_likelihood = np.mean(np.log(gaussian_distance(y_train, train_y_pred, var)))
test_log_likelihood = np.mean(np.log(gaussian_distance(y_test, test_y_pred, var)))

print("Train Log Likelihood: ", str(train_log_likelihood))
print("Test Log Likelihood: ", str(test_log_likelihood))


# Restore non-SE data for visualization
x_train, y_train = extend_x(train)

x = np.linspace(-1, 1, 1000)
x = np.vstack([x, np.ones(1000)])
x_se = squared_exponential(x.T, k, beta)[0]
y_pred = predict_y(x_se, w)

sigma = np.sqrt(compute_var(x_se, x_se, a, beta))

plt.scatter(x_train[0], y_train, c='k', label = 'Training data')
plt.plot(x[0], y_pred, c='tab:blue', label = 'Function prediction')

plt.fill_between(x[0], y_pred - std, y_pred + std, color='tab:blue', alpha=1/3)
plt.fill_between(x[0], y_pred - 2 * std, y_pred + 2 * std, color='tab:blue', alpha=1/3)
plt.fill_between(x[0], y_pred - 3 * std, y_pred + 3 * std, color='tab:blue', alpha=1/3)

plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.savefig('1d.png', bbox_inches='tight')
plt.clf()
```

2).

```
Train RSME:  0.08204602273485714
Test RSME:  0.1266499667226416
```

3).

```
Train Log Likelihood:  -0.04591945825929642
Test Log Likelihood:  -0.07191737612058605
```

4).
The SE transformation leads to the best results

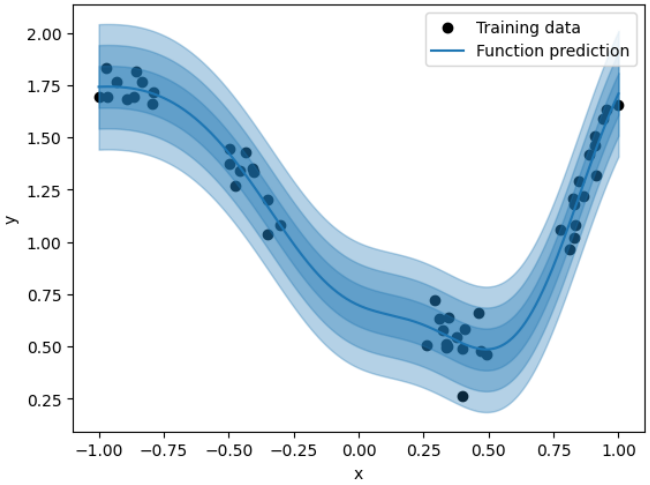

Figure 6: Training data and SE prediction

5).
The SE feature transformation resembles a Gaussian distribution.
α is like the mean μ as it gets subtracted from $\mathbf{X}_i$
β can be interpreted as $\frac{1}{\sigma^2}$, the multiplicative inverse of the variance σ^2

1e) Cross validation (4 Points)

Cross-validation is used to assess how well a model generalizes to data that is independent from the training set. The most common CV is K-Fold where instead of splitting the data into two, the data is split K-fold and then in each iteration some subsets (typically K-1) are used for training and the remaining subsets are used for validation.
In order to reduce variability of the partitioning K-Fold can be repeated multiple times with different partitioning and the evaluation results are combined (e.g. average). Another variant of CV is Leave-P-out which is basically N-Fold CV where N is the size of the training data and P subsets are used for validation. CV can further be enhanced by shuffling the data upfront and after each round.

Pros:

- Validates on multiple folds of data thus providing a measure for the model's generalization ability
- Can be used to balance out prediction labels if the dataset's class distribution is unbalanced
- Makes efficient use of the available data (does not require any additional validation data)
- Allows for hyperparameter optimization without needing to reserve some data exclusively for hyperparameter tuning

Cons:

- Requires that training and validation set are drawn from the same population and does therefor not work well for sequential data such as time series
- Computationally expensive because the model needs to be trained and evaluated multiple times

1f) Grid Search + Empirical Bayes (15 Points)

Task 2: Linear Classification (16 Points)

2a) Discriminative and Generative Models (4 Points)

The discriminant model learns $p(y|x)$ directly, or learns the category labels directly from the feature space. The common discriminant models are logistic regression.
The joint probability distribution $P(X,Y)$ is learned from the data, and then the probability distribution $P(Y|X)$ is derived from $P(Y|X) = \frac{P(X,Y)}{P(X)}$ as the model for prediction. This method represents the generative relationship between a given input X and the resulting output Y. Commonly used generative models include plain Bayesian, etc.
Generally, discriminant models are easier to learn as they only need to learn how to assign classes to inputs and do not learn the probability distribution of the data.

2b) Linear Discriminant Analysis (12 Points)

We decided to use a Maximum Likelihood discriminator[2] for the LDA prediction

```
X = np.loadtxt('dataSets/ldaData.txt')
y = np.empty(X.shape[0], dtype=int)
y[0:50] = 0
y[50:93] = 1
y[93:] = 2

class_colors = ['tab:red', 'tab:green', 'tab:blue']
X_classes = [X[0:50, :], X[50:93, :], X[93:, :]]

# Original dataset
for i in range(3):
   plt.scatter(X_classes[i][:, 0], X_classes[i][:, 1], color=class_colors[i],
       label='C'+str(i))

plt.title('Original Dataset')
plt.xlabel("x1")
plt.ylabel("x2")
plt.legend()
plt.savefig('2b_original.png', bbox_inches='tight')
plt.clf()


# LDA assumes normal distribution with equal covariance in all classes of X
def gaussian(data, mu, var):
   return np.array([1/np.sqrt(2*np.pi*var) * np.exp(-(x - mu)**2 / (2*var)) for x in data])

n_classes = 3
n, d = X.shape

# Scattering within and between classes
mean = np.mean(X, axis=0).reshape(1, d)

X_means = [np.mean(C, axis=0) for C in X_classes]
S_W = sum(sum(np.outer(x - mu, (x - mu).T) for x in C) for C, mu in zip(X_classes, X_means))

S_B = np.zeros((2,2))
for class_mean in X_means:
```

```
    S_B += (class_mean - mean) @ (class_mean - mean).T

# Computes eigenvalues and eigenvectors sorted in descending order
eigenvalues, eigenvectors = np.linalg.eig(np.linalg.inv(S_W) @ S_B)
w = eigenvectors[:,:1].T # take first dimension (linear projection)

# Linear projection of classes and their means and variances
linear_X_classes = [w @ C.T for C in X_classes]
linear_X_means = [np.mean(C) for C in linear_X_classes]
linear_X_variances = [np.var(C) for C in linear_X_classes]

# Assign X to the class with maximum likelihood discriminator
linear_X = np.concatenate([linear_X_classes[i][0] for i in range(n_classes)])
likelihood = [gaussian(linear_X, mean, variance) for mean, variance in zip(linear_X_means,
    linear_X_variances)]
y_pred = np.argmax(likelihood, axis=0)

c = [class_colors[y] for y in y_pred]

plt.scatter(X[:, 0], X[:, 1], c=c)

plt.title("LDA Classification")
plt.xlabel("x1")
plt.ylabel("x2")
plt.savefig('2b_lda.png', bbox_inches='tight')
plt.clf()

print('Misclassifications:', np.sum(y_pred != y))
```

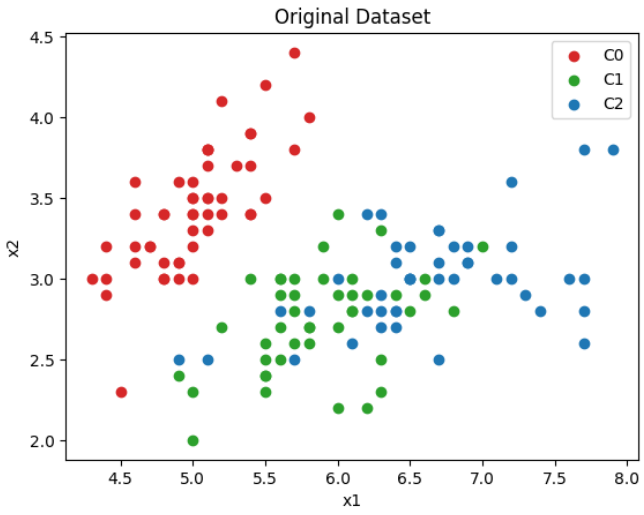

Figure 7: Original Dataset

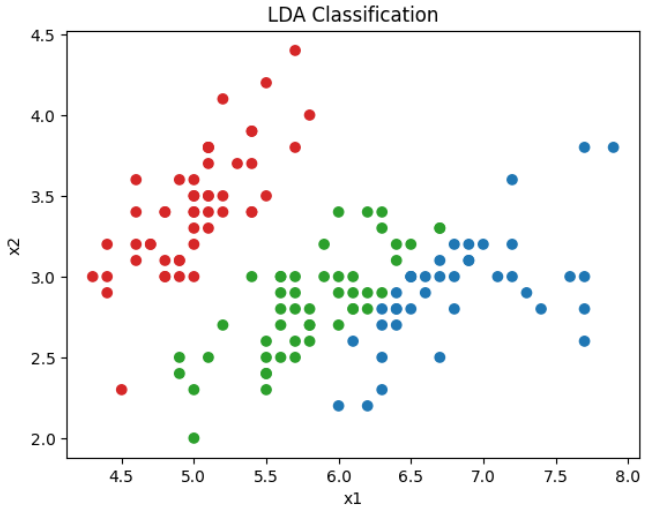

Figure 8: LDA Classification with ML Discriminator

The linear decision boundaries are easy to spot in the second plot. In total 25 samples are misclassified.

Task 3: Principal Component Analysis(28Points)

3a) Data Normalization (3 Points)

```
data = np.loadtxt('dataSets/iris.txt', delimiter=',')

X_original = data[:, :-1]
X_original_mean = np.mean(X_original, axis=0)
X_original_std = np.std(X_original, axis=0)

y = data[:, -1]

print('Mean (original): ', X_original_mean)
print('Variance (original): ', X_original_std)

# Normalization
X = (X_original - X_original_mean) / X_original_std

print('Mean (normalized): ', np.mean(X, axis=0))
print('Variance (normalized): ',np.var(X, axis=0))
```

```
Mean (original):  [5.84333333 3.054      3.75866667 1.19866667]
Variance (original):  [0.82530129 0.43214658 1.75852918 0.76061262]
Mean (normalized):  [-1.69031455e-15 -1.63702385e-15 -1.48251781e-15 -1.62314606e-15]
Variance (normalized):  [1. 1. 1. 1.]
```

Normalizing the data is important when features have different distributions to make sure that all features are on the same scale. Otherwise features with higher values by nature would have greater impact on the model than lower-value features for example but typically we don't want our model to have initial assumptions about the relevance of individual features as wrong biases can hinder the learning success.

3b) Principal Component Analysis (8 Points)

```python
def pca(X, D=None):
    # Principal Component Analysis with D components

    # Compute eigenvalues and eigenvectors of covariance matrix
    C = np.cov(X, rowvar=False)
    eigenvalues, eigenvectors = np.linalg.eig(C)

    # Fraction of the variance that eigenbasis with D components captures
    explained_vars = np.cumsum(eigenvalues) / np.sum(eigenvalues)

    if D is None:
        D = np.argmax(explained_vars >= 0.95) + 1

    # Take D eigenvectors with the highest eigenvalues as eigenbasis
    B = eigenvectors[:, :D]
    a = B.T @ X.T

    return B, a, D, explained_vars


# Compute PCA with 95% variance explained
B, a, D, explained_vars = pca(X)
print('At least {} components are required to explain 95% of the variance.'.format(D))

# Plot the explained variance
plt.plot(range(1, 5), explained_vars)
plt.xticks(range(1, 5))

# Plot the 95% boundary
xx, yy = np.meshgrid(np.linspace(1, 5, 1000), np.linspace(np.min(explained_vars), 1, 1000))
plt.contourf(xx, yy, yy >= 0.95, 2, cmap=cm.gray, alpha=0.5)

plt.title("Cumulative variance explained")
plt.xlabel("N. of components")
plt.ylabel("% variance explained")
plt.savefig('3b.png', bbox_inches='tight')
plt.clf()
```

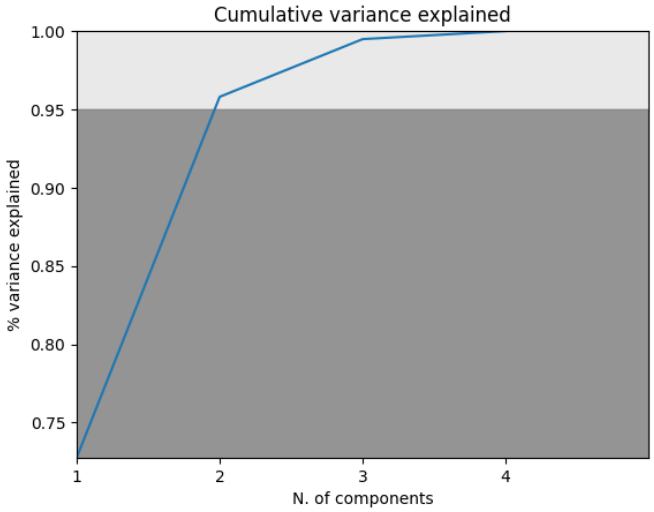

Figure 9: Cumulative variance explained by PCA

The PCA requires at least 2 components in order to explain 95% of the variance.

3c) Low Dimensional Space (6 Points)

```
class_colors = ['tab:red', 'tab:green', 'tab:blue']
c = [class_colors[int(i)] for i in y]

# Plot X w.r.t. the min. PCA basis that can explain 95% of the variance (2D)
# See 3b for the computation of a
plt.scatter(a[0], a[1], c=c)

plt.title("PCA with {} components".format(enough))
plt.xlabel("x1")
plt.ylabel("x2")
plt.savefig('3c.png', bbox_inches='tight')
plt.clf()
```

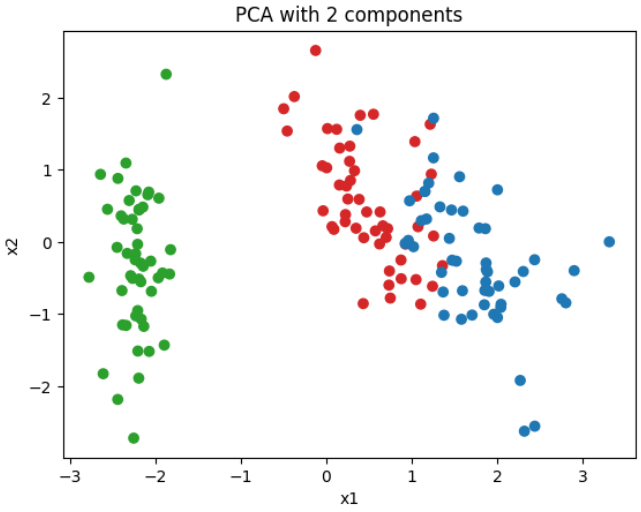

Figure 10: PCA with 2 components

3d) Projection to the Original Space (6 Points)

We considered the different normalization techniques[3] and decided to use range normalization

```python
def nrmse(X_pred, X):
    rmse = np.sqrt(np.mean((X - X_pred)**2, axis=0))

    # Normalize RMSE with the range
    y_max = np.max(X, axis=0)
    y_min = np.min(X, axis=0)
    return rmse / (y_max - y_min)

for D in range(1, 5):
    B, a, _, _ = pca(X, D)

    # X is reconstructed from a by applying reverse transformation
    X_reconstructed = (B @ a).T
    nrmse_D = nrmse(X_reconstructed, X)

    print('NRMSE (D={}): {}'.format(D, nrmse_D))
```

```
NRMSE (D=1): [0.10397936 0.16086196 0.03835783 0.08311765]
NRMSE (D=2): [0.06403369 0.0170341  0.03788015 0.08070068]
NRMSE (D=3): [0.00862221 0.00320869 0.03427903 0.02381893]
NRMSE (D=4): [1.42394418e-16 5.34157918e-17 1.40099782e-16 1.50415871e-16]
```

N. of components	x_1	x_2	x_3	x_4
1	0.10397936	0.16086196	0.03835783	0.08311765
2	0.06403369	0.0170341	0.03788015	0.08070068
3	0.00862221	0.00320869	0.03427903	0.02381893
4	1.42394418e-16	5.34157918e-17	1.40099782e-16	1.50415871e-16

Lower NRMSE values are an indicator for lower residual variance between the original and the reconstructed dataset. The results show more components lead to a lower reconstruction error which makes sense because more information from the original data is preserved by the PCA.

3e) Kernel PCA (5 Points)

KPCA is, in order to obtain better linear separability, the original data matrix is considered to be mapped to a new high-dimensional space, and then PCA is performed in the new high-dimensional space to reduce the dimensionality, but this processing requires a large number of computations, and the Kernel function can significantly reduce the amount of computations without affecting the effect of high-dimensional PCA.
The general process is:

a. Select kernel functions: linear kernel functions, polynomial kernel functions, Gaussian kernel functions, etc.

b. Construct the standard kernel matrix.

$$\widetilde{K} = K - \mathbf{1}_{1/n}K - K\mathbf{1}_{1/n} + \mathbf{1}_{1/n}K\mathbf{1}_{1/n}$$

c. Solve the feature vectors.

$$\widetilde{K}\alpha_i = \lambda_i\alpha_i$$

d. Mapping arbitrary original space data to the values of the space composed of principal components.

$$y_i = \sum_{i=1}^{n} \alpha_{ji}K(x, x_i), j = 1, ..., d$$

The KPCA will turn to have more computations if we have more data points than input variables.

References

[1] *Linear regression*. URL: https://en.wikipedia.org/wiki/Linear_regression.

[2] *LDA Discriminator*. URL: https://en.wikipedia.org/wiki/Linear_discriminant_analysis#Discrimination_rules.

[3] *RSME Normalization*. URL: https://en.wikipedia.org/wiki/Root-mean-square_deviation#Normalization.