Statistical Machine Learning - Exercise 3

TECHNISCHE UNIVERSITÄT DARMSTADT

Dominik Marino - 2468378, Pertami Kunz - 2380210
August 8, 2020

Contents

1 Linear Regression

1.1 1a)

1. Explain: What is the ridge coefficient and why do we use it?
   - The ridge coefficient minimize a penalized residual sum of squares and it is used in Ridge Regression (Variation of Linear Regression). We us a ridge coefficient when there are many correlated variables in a linear regression model. The coefficients can become determine and exhibit high variance.
   - Tutor solution:

     The ridge coefficient is a (typically small) positive number that determines how much Tikhonov regularization shall be used. It is used to improve numerical stability and to prevent overfitting

2. Derive the optimal model parameters by minimizing the squared error loss funciton
   Mean squared Error:

$$L(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(y - \hat{y}_i)^2$$

   Model:

$$\begin{aligned} y_i &= x^T w + \epsilon \\ 0 &= x^T w + \epsilon - y_i \\ &= x^T w - y_i \end{aligned}$$

   with loss function Mean squared Error:

$$\begin{aligned} \hat{w} &= argmin_\theta \frac{1}{2}||x^T w - y_i||^2 + \frac{\lambda}{2}||w||^2 \\ \nabla_w &\frac{1}{2}||x^T w - y_i||^2 + \frac{\lambda}{2}||w||^2 \end{aligned}$$

$$\begin{aligned} L(w, \lambda) &= \frac{1}{2}(Xw - y)^T(Xw - y) + \frac{\lambda}{2}w^T w \\ &= \frac{1}{2}(w^T X^T - y^T)(Xw - y)) + \frac{1}{2}w^T w \\ &= \frac{1}{2}(w^T X^T Xw - 2w^T X^T y + y^T y) + \frac{1}{2}w^T w \end{aligned}$$

   We suppose $w^T w \longrightarrow w^2$

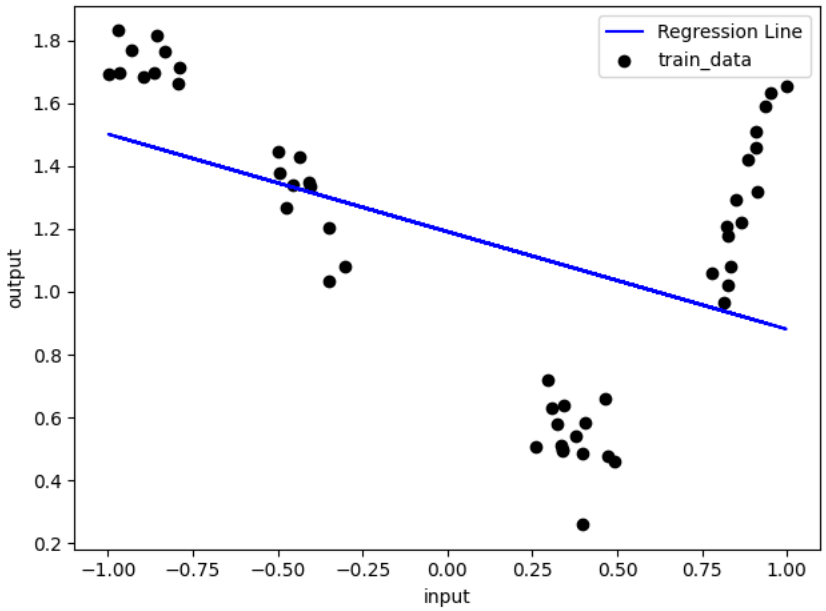
$$\begin{aligned} \frac{\partial L}{\partial w} &= X^T Xw - X^2 y + \lambda w \\ &= (X^T X + \lambda I)w - X^T y \\ 0 &= (X^T X + \lambda I)w - X^T y \\ x^T y &= (X^T X + \lambda I)w \\ X^T y &= (X^T X + \lambda I)w \\ w &= (X^T X + \lambda I)^{-1} X^T y \end{aligned}$$

3. Report the root mean squared error of the train and test data under your linear model with linear features
   Root Mean Squered Error (RMSE):

$$L(y, \hat{y}) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y - \hat{y}_i)^2}$$

	Train Set	TestSet
Error	0.41217801567361084	0.38428816992597886

4. Include a single plot that shows the training data as black dots and the predicted funciotn as a blue line




```python
def RMSD(predictions, targets):
    return np.sqrt(((predictions - targets) ** 2).mean())
```

Listing 1: Root Mean Squared Error

```python
def linear_regression(X, y, alpha, d = 1):
    X = polynomial_matrix(X, d)
    n, m = X.shape
    I = np.identity(m)
    return np.linalg.inv(X.T @ X + alpha * I) @ X.T @ y
```

Listing 2: Optimization Model

```python
def polynomial_matrix(X, d):
    x = []
    for i in range(0, d + 1):
        x.append(np.power(X, i))
    return np.asarray(x).T
```

Listing 3: Compute Polynomial Matrix

```python
def compute_prediction(X, w_hat, d=1):
    return polynomial_matrix(X, d) @ w_hat
```

Listing 4: Compute prediction y

```python
    w_hat = linear_regression(X_train, y_train, ridge_coefficient)
    y_pred_train = compute_prediction(X_train, w_hat)
    e_train = RMSD(y_pred_train, y_train)

    y_pred_test = compute_prediction(X_test, w_hat)
    e_test = RMSD(y_pred_test, y_test)

    w_hat_test = linear_regression(X_test, y_test, ridge_coefficient)
    y_pred_test = compute_prediction(X_test, w_hat_test)
    e_test_ = RMSD(y_pred_test, y_test)

    print("RMSD Train {}, \t RMSD Test {}, {}".format(e_train, e_test, e_test_))
    plt.figure()
    plt.plot(X_train, y_pred_train, c='b', label='Regression Line')
    show_data(X_train, y_train, 'train_data')
    plt.show()
```

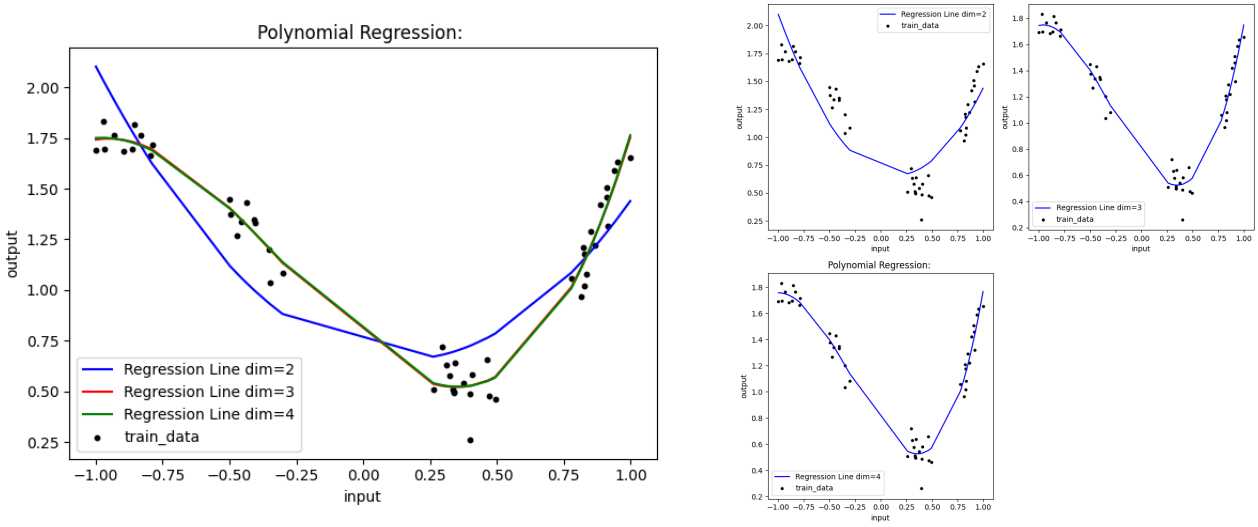
Listing 5: main

1.2 1b)

1. Report the root mean squared error of the training data and of the testing data under your model with polynomial features

degree	Trainingsdata	Testdata
2	0.2120144726596861	0.21687242714148738
3	0.08706821295481752	0.10835803719738046
4	0.08701261306638178	0.1066623982096429

2. Include a single plot that shows the training data as black dots and the predicted function as a blue line.



3. Why do we call this method linear regression despite using polynomials?
   - This model depends on a linear model. We are using polynomial but is has no influence of our regression coefficient. The polynomials does not change the model but can be described as a multiple combination of linear regression coefficients.
   - Tutor solution:

     We still call this linear regression because the data model $y_i = \mathbf{p}(x_i)^T \mathbf{w} + \epsilon_i$ remains linear with respect to the parameter $\mathbf{w}$. This is particularly important for Bayesian linar regression.

```python
#Root Mean Squared Error
def RMSD(predictions, targets):
    return np.sqrt(((predictions - targets) ** 2).mean())

def show_data(x, y, name):
    plt.scatter(x, y, color='k', label=name, s=10)
    plt.legend()
    plt.xlabel('input')
    plt.ylabel('output')

def linear_regression(X, y, alpha, d = 1):
    X = polynomial_matrix(X, d)
    n, m = X.shape
    I = np.identity(m)
    return np.linalg.inv(X.T @ X + alpha * I) @ X.T @ y

def compute_prediction(X, w_hat, d=1):
    return polynomial_matrix(X, d) @ w_hat

def polynomial_matrix(X, d):
    x = []
    for i in range(0, d + 1):
        x.append(np.power(X, i))
    return np.asarray(x).T

```

Listing 6: Same like above

```python
    degrees = [2, 3, 4]
    color_map = ['b', 'b', 'b']

```

```
    plt.subplots_adjust(wspace=0.5, hspace=0.5)
    plt.figure(figsize=(10, 10))

    i = 221
    show_data(X_train, y_train, 'train_data')
    for d, color in zip(degrees, color_map):
        w_hat = linear_regression(X_train, y_train, ridge_coefficient, d)
        y_pred = compute_prediction(X_train, w_hat, d)
        error = RMSD(y_pred, y_train)
        print('1b) Error: ', error)

        sorted_zip = sorted(zip(X_train, y_pred))
        x, y = zip(*sorted_zip)
        plt.subplot(i)
        i = i + 1
        show_data(X_train, y_train, 'train_data')
        plt.plot(x, y, c=color, label='Regression Line dim=' + str(d))
        plt.legend()
    plt.title(label='Polynomial Regression: ')
    plt.show()

```

Listing 7: main

1.3 1c)

1. For each polynomial degree, report the average train, validation and test RMSEs among all folds

Dimension	Training	Validation	Test
2	0.20943990135161356	0.22488505597839778	0.21835094011192718
3	0.08620813857069498	0.09271100111785265	0.10927671570049995
4	0.0854725162035436	0.09841566883354019	0.10867173876433567

2. Explain: Do the resulting numbers meet your expectations? Why (not)?
   - The resulting numbers meet my expectation, because if we look up to the graphs in 1b, we trained our model over all the data. With a higher dimension we get a smoother line to fit the data better and it leads to a smaller error. We see also with a dimension of 4 that the Trainingserror gets smaller and the error of the validation set and trainingsset get higher. So it may lead to overfit.
   - Tutor solution:

     As the polynomial degree increases, we increase the expensiveness of the model. Therefore, we expect the train RMSE to decrease and the validation RMSE to increase eventually due to overfitting. The results match our expectations because the train RMSE consistently decreases as we increase the degree of the polynomial, but the validation RMSE starts to increase again as we increase the degree from 3 to 4

3. Which polynomial degree should be chosen for the given data? Why?
   - Wie should choose a degree of 3. The average errors in Validation and Test is the error with the dimension 3 the smallest value. With a dimension of 4 the Trainingserror gets at its lowest, but we dont matter the error. We are more interested in the Validation- and Testerror. We also see, that the error of the training-set gets lower with a higher dimension meanwhile the validataion- and test-set-error gets an higher error. Therefore, we can support our assumption that with a higher dimension it leads to overfit.
   - Tutor solution:

     Based on the validation results, we should choose 3 as the polynomial degree because it comes with the lowest validation RMSE. Although train RMSE and test RMSE are lower for 4, the train RMSE is prone to overfitting and the test RMSE must not be used for any model selection. Using the test data for model selection violates the 'golden rule' of machine learning.

```
    def cross_validation(X_data, y_data, i, size=0.2):
    train_size = len(y_data) * size
    X_val, y_test, X_train, y_val = [], [], [], []

    min = round(train_size * i)
    max = round(train_size * i + train_size)
    for idx, (X, y) in enumerate(zip(X_data, y_data)):
       if(idx >= min and idx < max):
           X_val.append(X)
           y_test.append(y)
       else:
           X_train.append(X)
```

```
            y_val.append(y)

    return np.asarray(X_train), np.asarray(X_val), np.asarray(y_val), np.asarray(y_test)

```

Listing 8: Cross-Validation: Seperate the data into a training and validation set

```
#Root Mean Squared Error
def RMSD(predictions, targets):
    return np.sqrt(((predictions - targets) ** 2).mean())

def show_data(x, y, name):
    plt.scatter(x, y, color='k', label=name, s=10)
    plt.legend()
    plt.xlabel('input')
    plt.ylabel('output')

def linear_regression(X, y, alpha, d = 1):
    X = polynomial_matrix(X, d)
    n, m = X.shape
    I = np.identity(m)
    return np.linalg.inv(X.T @ X + alpha * I) @ X.T @ y

def compute_prediction(X, w_hat, d=1):
    return polynomial_matrix(X, d) @ w_hat

```

```
    error_2, error_3, error_4 = [], [], []
    for i in range(5):      #5 subsets
        X_train, X_val, y_train, y_val = cross_validation(train_data[:, 0], train_data[:, 1], i)
        for d in degrees:       #degrees = [2, 3, 4]
            w_hat = linear_regression(X_train, y_train, ridge_coefficient, d)
            y_pred = compute_prediction(X_train, w_hat, d)
            error_train = RMSD(y_pred, y_train)

            y_pred = compute_prediction(X_val, w_hat, d)
            error_val = RMSD(y_pred, y_val)

            y_pred = compute_prediction(X_test, w_hat, d)
            error_test = RMSD(y_pred, y_test)

            if(d == 2):
                error_2.append([error_train, error_val, error_test])
            elif(d == 3):
                error_3.append([error_train, error_val, error_test])
            else:
                error_4.append([error_train, error_val, error_test])

    #Calculate Average of
    error_2 = np.asarray(error_2)
    error_3 = np.asarray(error_3)
    error_4 = np.asarray(error_4)
    sets = ['Train', 'Val', 'Test']
    for i, s in zip(range(error_2.shape[1]), sets):
        print("Average value of Error: {} -->  dim2: {}, dim3: {}, dim4: {}".format(s, error_2[:, i].mean(),error_3
     [:, i].mean(), error_4[:, i].mean()))

```

Listing 9: Compute Error of train-, validation- and test-set

1.4 1d)

1. State the posterior distribution of the model parameters p(w | X, y) (no derivation required)

$$p(w \mid X, y) = p(y \mid X, w, \beta)p(w \mid \alpha)$$
$$p(w \mid X, y) = \mathcal{N}(y \mid X, \beta^{-1})\mathcal{N}(0, \alpha^{-1})$$

- Tutor solution:

$$p(w \mid X, y) = \mathcal{N}(\mu_n, \Lambda_n^{-1})$$
$$\mu_n = \sigma^{-2}\Lambda_n^{-1}X^T y$$
$$\Lambda_n^{-1} = \sigma^{-2}X^T X + \lambda I$$

2. State the predictive distribution $p(y_* \mid X_*, X, y)$ (no derivation required)

$$p(y_* \mid X_*, X, y) = \int p(y_* \mid X_*, \theta)p(\theta, X, y)d\theta$$

Predictive distribution:

$$p(y_* \mid X_*, X, y) = N(y_* \mid \mu(X_*, \sigma^2(X_*)))$$
$$\mu(X_*) = \Phi^T(X_*)(\frac{\alpha}{\beta}I + \Phi\Phi^T)^{-1}\Phi^T y$$
$$\sigma^2(X_*) = \frac{1}{\beta} + \Phi^T(X_*)(\alpha I + \beta\Phi\Phi^T)^{-1}\Phi(X_*)$$

- Tutor solution:

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

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

Train	Test
0.4121779259165973	0.38434085452132943

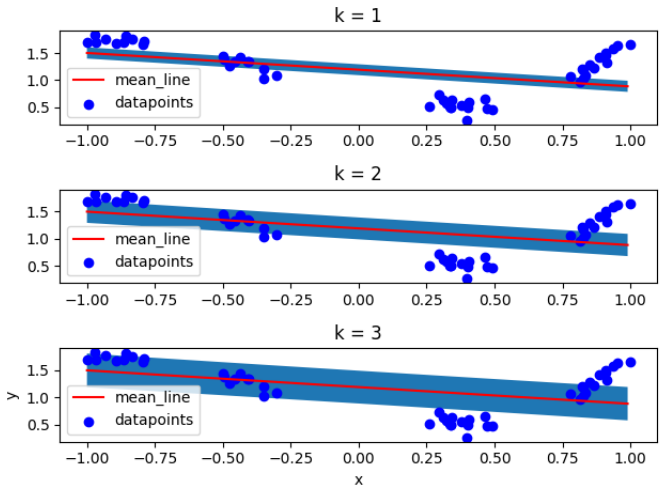
4. Report the average log-likelihood of the train and test data under your Bayesian model

Train	Test
-87.70894140971299	-90.76197537573506

- Tutor Solution:

Train	Test
-6.83469956991346	-5.774748828572732

5. Include a single 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 (you can use matplotlib's fill between function for that)



6. Explain the differences between linear regression and Bayesian linear regression
   - In Bayesian regression, we look for a predictive probability or the sampling to look for posterior probability distribution of the model. So if x is a new point, we look up for the probability of the y-value corresponding to the given x. In linear regression we are using a linear function model to estimate the unknown parameters form its data.
   - Tutor solution:

     In linear regression, we compute the optimal value for the parameters $\hat{\mathbf{w}}$ by setting the gradient of the squared error loss function to zero. Problematically, this is equivalent to the maximum likelihood point estimate under Gaussian assumptions. However, in Bayesian linear regression, instead of the maximum likelihood estimate, we compute the full posterior distribution of the parameters $\mathbf{w}$ using Bayes rule. To the end, we define the data likelihood $p(\mathcal{D}) = \int p(\mathcal{D} \mid \mathbf{w}) p(\mathbf{w}) dw$. Main take-away: linear regression computes a single vector for $\mathbf{w}$, Bayesian linear regression computes a full probability distribution for $\mathbf{w}$

```
1
2  def log_likelihood(p):
3      return np.sum(np.log(p)) / len(p)
4
5  def likelihood(mu, sigma, x):
6      factor = 1 / np.sqrt(2*np.pi*sigma**2)
7      return factor * np.exp(-0.5 * ((x - mu)/ sigma)**2)
8
9
10 def baysian_linear_regression(phi, y, alpha, beta):
11     print(alpha, beta, " alpha und beta")
12     n, m = phi.shape
13     I = np.identity(m)
14
15     mean_matrix = np.linalg.inv((alpha / beta) * I + phi.T @ phi) @ phi.T @ y
16     variance_matrix = np.linalg.inv(alpha*I + beta * phi.T @ phi)
17
18     def predictive_mean(x):
19         return x.T @ mean_matrix
20
21     def predictive_variance(x):
22         return 1/beta + x.T @ variance_matrix @ x
23
24     means = []
25     variance = []
26     likeli = []
27     for x in phi:
28         m = predictive_mean(x)
29         means.append(m)
30         v = predictive_variance(x)
31         variance.append(v)
32         likeli.append(likelihood(m, np.sqrt(v), x))
33     std = np.asarray(np.sqrt(variance))
34     means = np.asarray(means)
```

```
    print("Average Error: ", RMSD(means, y))
    print("average Likelihood: ",  log_likelihood(np.asarray(likeli)[:, 1]))

    c = [1, 2, 3]
    X = np.arange(-1, 1, 0.01)
    X = np.c_[np.ones(len(X)), X]

    means = []
    variances = []
    for n in X:
        means.append(predictive_mean(n))
        variances.append(predictive_variance(n))
    means = np.asarray(means)
    std = np.sqrt(np.asarray(variances))

    fig, axs = plt.subplots(len(c))
    for idx, k in enumerate(c):
        axs[idx].plot(X[:, 1], means, label='mean_line', color='r')
        axs[idx].fill_between(X[:, 1], (means - k * std), (means + k * std))
        axs[idx].scatter(phi[:, 1], y, label='datapoints', c='b')
        axs[idx].set_title('k = {}'.format(k))
        axs[idx].legend()
        plt.xlabel('x')
        plt.ylabel('y')
        plt.legend()
    plt.show()


```

Listing 10: Bayesian Linear Regression

```
def calc_error_of_train(phi_train, phi_test, y_train, y_test, alpha, beta):
    n, m = phi_train.shape
    I = np.identity(m)

    mean_matrix = np.linalg.inv((alpha / beta) * I + phi_train.T @ phi_train) @ phi_train.T @ y_train
    variance_matrix = np.linalg.inv(alpha * I + beta * phi_train.T @ phi_train)

    def predictive_mean(x):
        return x.T @ mean_matrix

    def predictive_variance(x):
        return 1 / beta + x.T @ variance_matrix @ x

    means = []
    variance = []
    likeli = []
    for x in phi_test:
        m = predictive_mean(x)
        means.append(m)
        v = predictive_variance(x)
        variance.append(v)
        likeli.append(likelihood(m, np.sqrt(v), x))
    std = np.asarray(np.sqrt(variance))
    means = np.asarray(means)
    print("1e) Average Error: ", RMSD(means, y_test))
    print("1e) average Likelihood: ", log_likelihood(np.asarray(likeli)[:, 1]))


```

Listing 11: Evaluation under Trained Model

```
    ridge_coefficient = 0.01
    X_train = train_data[:, 0]
    y_train = train_data[:, 1]
    X_test = test_data[:, 0]
    y_test = test_data[:, 1]
    baysian_linear_regression(np.c_[np.ones(len(X_train)), X_train], y_train, ridge_coefficient, 1/0.01)
    baysian_linear_regression(np.c_[np.ones(len(X_test)), X_test], y_test, ridge_coefficient, 1/0.01)
    calc_error(np.c_[np.ones(len(X_train)), X_train], np.c_[np.ones(len(X_test)), X_test], y_train, y_test,
     ridge_coefficient, 1 / 0.01)

```

Listing 12: main

1.5 1e)

1. Report the RMSE of the train and test data under your Bayesian model with SE features. (1)

Train	Test
0.08188803394360888	0.16297037244258858 / sol: 0.138874367..

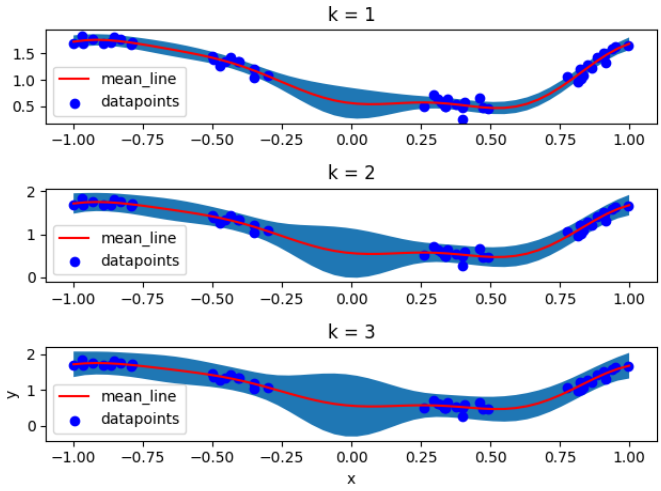
2. Report the average log-likelihood of the train and test data under your Bayesian model with SE features.

Train	Test
-38.05692344875588	-23.936399413849045

- Tutor solution:

Train	Test
1.013735727793537	0.57764976548710

3. Include a single 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 (you can use matplotlib's fill between function for that



4. How can SE features be interpreted from a statisticians point of view? What are α and β in that context?
   - it shows us the relationship between the independent variable x and the dependent variable y. In task 1b) the polynomial regression is a nonlinear model to the data, like a statistical estimation problem. The polynomial regression is a special case of multiple linear regression. With squared exponential features we can create a much smoother multiple linear regression.
     * α: distrust of a strong β
     * β: precision of the noise
   - Tutor solution:

     Squared exponential featrues are equivalent to Gaussian basis function where α represents the mean of these functions and β is the precision, i.e. inverse variance.

```
  def squad_exponentional_matrix(k, X, beta = 10):
    phi = []
    for i in range(k):
        alpha = np.ones(len(X)) * i * 0.1 - 1
        vec = np.exp(-0.5 * beta * (X - alpha) ** 2)
        phi.append(vec)
    return np.asarray(phi).T

```

Listing 13: create SE matrix

```
def squad_exponentional_matrix(k, X, beta = 10):
    phi = []
    for i in range(k):
        alpha = np.ones(len(X)) * i * 0.1 - 1
        vec = np.exp(-0.5 * beta * (X - alpha) ** 2)
        phi.append(vec)
    return np.asarray(phi).T

def log_likelihood(p):
    return np.sum(np.log(p)) / len(p)

def likelihood(mu, sigma, x):
    factor = 1 / np.sqrt(2*np.pi*sigma**2)
    return factor * np.exp(-0.5 * ((x - mu)/ sigma)**2)

def baysian_regression(phi, X_data, y, alpha, beta):
    n, m = phi.shape
    I = np.identity(m)
    mean_matrix = np.linalg.inv((alpha / beta) * I + phi.T @ phi) @ phi.T @ y
    variance_matrix = np.linalg.inv(alpha * I + beta * phi.T @ phi)

    def predictive_mean(x):
        return x.T @ mean_matrix

    def predictive_variance(x):
        return 1 / beta + x.T @ variance_matrix @ x

    means = []
    variances = []
    likeli = []
    for row in phi:
        m = predictive_mean(row)
        means.append(m)
        v = predictive_variance(row)
        variances.append(v)
        likeli.append(likelihood(m, np.sqrt(v), row))

    means = np.asarray(means)
    print("1e) Average Error: ", RMSD(means, y))
    print("1e) average Likelihood: ",  log_likelihood(np.asarray(likeli)[:, 1]))

    c = [1, 2, 3]
    X_val = np.linspace(-1, 1, len(y))
    X_val = squad_exponentional_matrix(20, X_val)

    means = []
    variances = []
    for x in X_val:
        means.append(predictive_mean(x))
        variances.append(predictive_variance(x))

    means = np.asarray(means)
    std = np.sqrt(np.asarray(variances))
```

```
    X = np.linspace(-1, 1, len(y))

    fig, axs = plt.subplots(len(c))
    for idx, k in enumerate(c):
        axs[idx].plot(X, means, label='mean_line', color='r')
        axs[idx].fill_between(X, (means - k * std), (means + k * std))
        axs[idx].scatter(X_data, y, label='datapoints', c='b')
        axs[idx].set_title('k = {}'.format(k))
        axs[idx].legend()
        plt.xlabel('x')
        plt.ylabel('y')
        plt.legend()
    plt.show()
```

Listing 14: bayesian regression

```
    phi_train = squad_exponentional_matrix(20, X_train)
    phi_test = squad_exponentional_matrix(20, X_test)
    calc_error_of_train(phi_train, phi_test, y_train, y_test, ridge_coefficient, 1/0.01)
    baysian_regression(phi_train, X_train, y_train, ridge_coefficient, 1/0.01)
    baysian_regression(phi_test, X_test, y_test, ridge_coefficient, 1/0.01)
```

Listing 15: main

1.6 1f)

1. What is the difference between the marginal likelihood p(y | X) and the likelihood p(y | X, w)? (1)
   - The marginal likelihood provides a priciple and automatic way of model comparison. So it is used to select between models. For each model we take all the posterior function to reject sampling. Likelihood describes the density of the estimation of the data.
   - tutor solution:

     The marginal likelihood p(y| X) does not depend on the model parameters w because the have bin marginalized out via integration. Intuitively, the marginal likelihood represents the probability of the data averaged over possible model parameters w, as specified by the prior p(w). The likelihood p(y | X, w) depents on a specific w

2. For each β, report RMSE and average log-likelihood of the train and test data and the log-marginal likelihood

beta	RMSE Train	RMSE Test
1	0.08981902717812963	0.12423823423981933 / sol: 0.1083688207135218
10	0.08241791522223116	0.16897517789411245 / sol: 0.1425781435565231
100	0.08188803394360888	0.16297037244258858 / sol: 1.2300696160551463

β	log-likelihood (Train)	log-likelihood (Test)
1	0.9616249230979097	0.8030792476697927
10	1.013119934600136	0.5605016434925285
100	1.0401611897246368	-0.4814402200628525

β	log-marginal likelihood
1	27.170162750726966
10	11.915076107400346
100	-14.575065157319287

3. For each beta, include a single plot that shows the training data as black dots, the mean of the predictive distri- bution as blue line and 1, 2 and 3 standard deviations of the predictive distribution in shades of blue (you can use matplotlib's fill between function for that

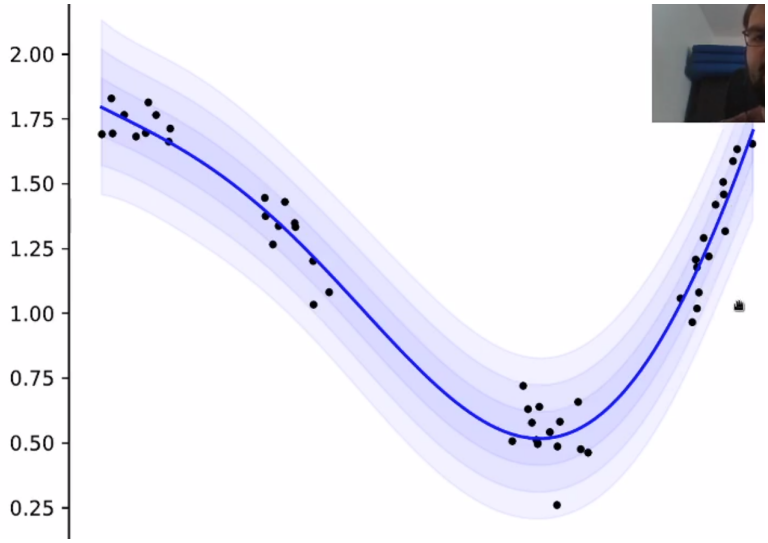


Figure 1: $\beta = 1$

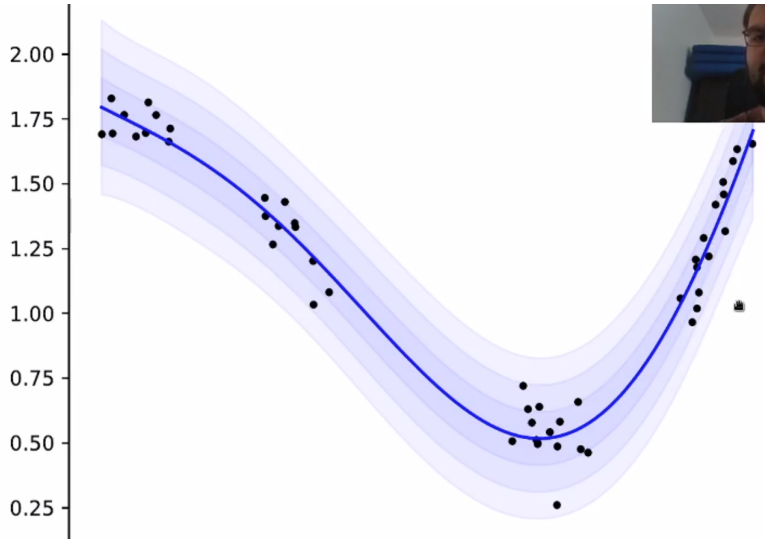


Figure 2: $\beta = 10$

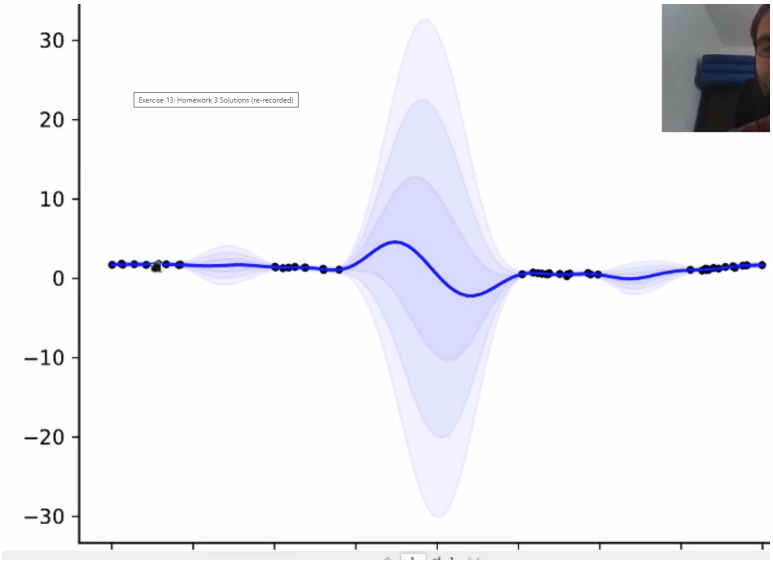


Figure 3: β = 100

4. According to the grid search, which value for β is the best? Why?
   - According our result, i would take $\beta = 100$, because it covers better the train and test data. The Error is acceptable.
   - Tutor solution:

     Accourding to the grid search, $\beta = 1$ is the best value because it yielded the highest log-marginal likelihood

5. Compare the log-marginal likelihood values to the average train and test log-likelihood values. What do you observe? Is the log-marginal likelihood a 'good' score function compared to the train log-likelihood?

- Tutor solution:

  The average train log-likelihood consistently increases as β increases, whereas the log-marginal likelihood decreases as β increases. In particular, consistent with the log-marginal likelihood, the average test log-likelihood also decreases as β increases. Therefore, to prevent overfitting, the log-marginal likelihood can be used for model selection without looking at the test data. This property makes the log-marginal likelihood a good score function.

```
    beta = [1, 10, 100]
    for b in beta:
        baysian_regression(phi_train, X_train, y_train, 0.01, b)
        baysian_regression(phi_test, X_test, y_test, 0.01, b)
        calc_error_of_train(phi_train, phi_test, y_train, y_test, ridge_coefficient, b)
```

Listing 16: main

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

#Root Mean Squared Error
def RMSD(predictions, targets):
    return np.sqrt(((predictions - targets) ** 2).mean())

def show_data(x, y, name):
    plt.scatter(x, y, color='k', label=name, s=10)
    plt.legend()
    plt.xlabel('input')
    plt.ylabel('output')

def linear_regression(X, y, alpha, d = 1):
    X = polynomial_matrix(X, d)
    n, m = X.shape
    I = np.identity(m)
    return np.linalg.inv(X.T @ X + alpha * I) @ X.T @ y

def compute_prediction(X, w_hat, d=1):
    return polynomial_matrix(X, d) @ w_hat
```

```

def polynomial_matrix(X, d):
    x = []
    for i in range(0, d + 1):
        x.append(np.power(X, i))
    return np.asarray(x).T

def cross_validation(X_data, y_data, i, size=0.2):
    train_size = len(y_data) * size
    X_val, y_test, X_train, y_val = [], [], [], []

    min = round(train_size * i)
    max = round(train_size * i + train_size)
    for idx, (X, y) in enumerate(zip(X_data, y_data)):
       if(idx >= min and idx < max):
           X_val.append(X)
           y_test.append(y)
       else:
           X_train.append(X)
           y_val.append(y)

    return np.asarray(X_train), np.asarray(X_val), np.asarray(y_val), np.asarray(y_test)

def log_likelihood(p):
    return np.sum(np.log(p)) / len(p)

def likelihood(mu, sigma, x):
    factor = 1 / np.sqrt(2*np.pi*sigma**2)
    return factor * np.exp(-0.5 * ((x - mu)/ sigma)**2)


def baysian_linear_regression(phi, y, alpha, beta):
    print(alpha, beta, " alpha und beta")
    n, m = phi.shape
    I = np.identity(m)

    mean_matrix = np.linalg.inv((alpha / beta) * I + phi.T @ phi) @ phi.T @ y
    variance_matrix = np.linalg.inv(alpha*I + beta * phi.T @ phi)

    def predictive_mean(x):
        return x.T @ mean_matrix

    def predictive_variance(x):
        return 1/beta + x.T @ variance_matrix @ x

    means = []
    variance = []
    likeli = []
    for x in phi:
        m = predictive_mean(x)
        means.append(m)
        v = predictive_variance(x)
        variance.append(v)
        likeli.append(likelihood(m, np.sqrt(v), x))
    std = np.asarray(np.sqrt(variance))
    means = np.asarray(means)

    print("Average Error: ", RMSD(means, y))
    print("average Likelihood: ",  log_likelihood(np.asarray(likeli)[:, 1]))

    c = [1, 2, 3]
    X = np.arange(-1, 1, 0.01)
    X = np.c_[np.ones(len(X)), X]

    means = []
    variances = []
    for n in X:
        means.append(predictive_mean(n))
        variances.append(predictive_variance(n))
    means = np.asarray(means)
    std = np.sqrt(np.asarray(variances))

    fig, axs = plt.subplots(len(c))
    for idx, k in enumerate(c):
        axs[idx].plot(X[:, 1], means, label='mean_line', color='r')
        axs[idx].fill_between(X[:, 1], (means - k * std), (means + k * std))
        axs[idx].scatter(phi[:, 1], y, label='datapoints', c='b')
        axs[idx].set_title('k = {}'.format(k))
        axs[idx].legend()
        plt.xlabel('x')
```

```
        plt.ylabel('y')
        plt.legend()
    plt.show()

def baysian_regression(phi, X_data, y, alpha, beta):
    n, m = phi.shape
    I = np.identity(m)
    mean_matrix = np.linalg.inv((alpha / beta) * I + phi.T @ phi) @ phi.T @ y
    variance_matrix = np.linalg.inv(alpha * I + beta * phi.T @ phi)

    def predictive_mean(x):
        return x.T @ mean_matrix

    def predictive_variance(x):
        return 1 / beta + x.T @ variance_matrix @ x

    means = []
    variances = []
    likeli = []
    for row in phi:
        m = predictive_mean(row)
        means.append(m)
        v = predictive_variance(row)
        variances.append(v)
        likeli.append(likelihood(m, np.sqrt(v), row))

    means = np.asarray(means)
    print("1e) Average Error: ", RMSD(means, y))
    print("1e) average Likelihood: ",  log_likelihood(np.asarray(likeli)[:, 1]))

    c = [1, 2, 3]
    X_val = np.linspace(-1, 1, len(y))
    X_val = squad_exponentional_matrix(20, X_val)

    means = []
    variances = []
    for x in X_val:
        means.append(predictive_mean(x))
        variances.append(predictive_variance(x))

    means = np.asarray(means)
    std = np.sqrt(np.asarray(variances))

    X = np.linspace(-1, 1, len(y))

    fig, axs = plt.subplots(len(c))
    for idx, k in enumerate(c):
        axs[idx].plot(X, means, label='mean_line', color='r')
        axs[idx].fill_between(X, (means - k * std), (means + k * std))
        axs[idx].scatter(X_data, y, label='datapoints', c='b')
        axs[idx].set_title('k = {}'.format(k))
        axs[idx].legend()
        plt.xlabel('x')
        plt.ylabel('y')
        plt.legend()
    plt.show()

def squad_exponentional_matrix(k, X, beta = 10):
    phi = []
    for i in range(k):
        alpha = np.ones(len(X)) * i * 0.1 - 1
        vec = np.exp(-0.5 * beta * (X - alpha) ** 2)
        phi.append(vec)
    return np.asarray(phi).T

def doBayesianRegression(phi, X_data, y, alpha, beta, sigma = 0.1):
    n, m = phi.shape
    I = np.identity(m)

    mu = sigma**-2 * np.linalg.inv((alpha/beta)*I) @ phi.T @ y
    sig = sigma**-2 * phi.T @ phi + (alpha / beta) * I

    def logMarginalLikelihood(x):
        return ((phi.shape[1] + 1) / 2) * np.log(alpha/beta) \
               - (phi.shape[0] / 2) * np.log(sig ** 2) \
               - 0.5 * np.linalg.norm(y - x @ mu)**2 / sig**2 \
               + 0.5 * (alpha/beta) * mu.T @ mu \
               - 0.5 * np.log(sig) \
               - 0.5 * phi.shape[0]*np.log(2*np.pi)

```

```
182
183 def calc_error_of_train(phi_train, phi_test, y_train, y_test, alpha, beta):
184     n, m = phi_train.shape
185     I = np.identity(m)
186
187     mean_matrix = np.linalg.inv((alpha / beta) * I + phi_train.T @ phi_train) @ phi_train.T @ y_train
188     variance_matrix = np.linalg.inv(alpha * I + beta * phi_train.T @ phi_train)
189
190     def predictive_mean(x):
191         return x.T @ mean_matrix
192
193     def predictive_variance(x):
194         return 1 / beta + x.T @ variance_matrix @ x
195
196     means = []
197     variance = []
198     likeli = []
199     for x in phi_test:
200         m = predictive_mean(x)
201         means.append(m)
202         v = predictive_variance(x)
203         variance.append(v)
204         likeli.append(likelihood(m, np.sqrt(v), x))
205     std = np.asarray(np.sqrt(variance))
206     means = np.asarray(means)
207     print("1e) Average Error: ", RMSD(means, y_test))
208     print("1e) average Likelihood: ", log_likelihood(np.asarray(likeli)[:, 1]))
209
210
211 def main():
212     #inizialization
213     ridge_coefficient = 0.01
214     train_data = np.genfromtxt('dataSets/lin_reg_train.txt', delimiter=' ')
215     test_data = np.loadtxt('dataSets/lin_reg_test.txt', delimiter=' ')
216
217     X_train = train_data[:, 0] #np.c_[np.ones(len(train_data[:, 0])), train_data[:, 0]]
218     y_train = train_data[:, 1]
219
220     X_test = test_data[:, 0] #np.c_[np.ones(len(test_data[:, 0])), test_data[:, 0]]
221     y_test = test_data[:, 1]
222
223     """ 1a) """
224     w_hat = linear_regression(X_train, y_train, ridge_coefficient)
225     y_pred_train = compute_prediction(X_train, w_hat)
226     e_train = RMSD(y_pred_train, y_train)
227
228     y_pred_test = compute_prediction(X_test, w_hat)
229     e_test = RMSD(y_pred_test, y_test)
230
231     w_hat_test = linear_regression(X_test, y_test, ridge_coefficient)
232     y_pred_test = compute_prediction(X_test, w_hat_test)
233     e_test_ = RMSD(y_pred_test, y_test)
234
235     print("RMSD Train {}, \t RMSD Test {}, {}".format(e_train, e_test, e_test_))
236
237     plt.figure()
238     plt.plot(X_train, y_pred_train, c='b', label='Regression Line')
239     show_data(X_train, y_train, 'train_data')
240     plt.show()
241
242     """ 1b) """
243     degrees = [2, 3, 4]
244     color_map = ['b', 'b', 'b']
245
246     plt.subplots_adjust(wspace=0.5, hspace=0.5)
247     plt.figure(figsize=(10, 10))
248
249     i = 221
250     show_data(X_train, y_train, 'train_data')
251     for d, color in zip(degrees, color_map):
252         w_hat = linear_regression(X_train, y_train, ridge_coefficient, d)
253         y_pred = compute_prediction(X_train, w_hat, d)
254         error = RMSD(y_pred, y_train)
255         print('1b) Error: ', error)
256
257         sorted_zip = sorted(zip(X_train, y_pred))
258         x, y = zip(*sorted_zip)
259         plt.subplot(i)
260         i = i + 1
261         show_data(X_train, y_train, 'train_data')
```

```
        plt.plot(x, y, c=color, label='Regression Line dim=' + str(d))
        plt.legend()
    plt.title(label='Polynomial Regression: ')
    plt.show()

    error_2, error_3, error_4 = [], [], []
    """ 1c) """
    for i in range(5):
        X_train, X_val, y_train, y_val = cross_validation(train_data[:, 0], train_data[:, 1], i)
        for d in degrees:
            w_hat = linear_regression(X_train, y_train, ridge_coefficient, d)
            y_pred = compute_prediction(X_train, w_hat, d)
            error_train = RMSD(y_pred, y_train)

            y_pred = compute_prediction(X_val, w_hat, d)
            error_val = RMSD(y_pred, y_val)

            y_pred = compute_prediction(X_test, w_hat, d)
            error_test = RMSD(y_pred, y_test)

            if(d == 2):
                error_2.append([error_train, error_val, error_test])
            elif(d == 3):
                error_3.append([error_train, error_val, error_test])
            else:
                error_4.append([error_train, error_val, error_test])

            #print("Dim: {}, i: {} -->  Error Train: {}, Error Val: {}, Error Test: {}".format(d, i, error_train,
     error_val, error_test))
#
    #Calculate Average of
    error_2 = np.asarray(error_2)
    error_3 = np.asarray(error_3)
    error_4 = np.asarray(error_4)
    set = ['Train', 'Val', 'Test']
    for i, set in zip(range(error_2.shape[1]), set):
        print("Average value of Error: {} -->  dim2: {}, dim3: {}, dim4: {}".format(set, error_2[:, i].mean(),
                                                                                    error_3[:, i].mean(), error_4[:, i].
     mean()))

    """ 1d) """
    ridge_coefficient = 0.01
    X_train = train_data[:, 0]
    y_train = train_data[:, 1]
    X_test = test_data[:, 0]
    y_test = test_data[:, 1]
    baysian_linear_regression(np.c_[np.ones(len(X_train)), X_train], y_train, ridge_coefficient, 1/0.01)
    baysian_linear_regression(np.c_[np.ones(len(X_test)), X_test], y_test, ridge_coefficient, 1/0.01)
    calc_error_of_train(np.c_[np.ones(len(X_train)), X_train], np.c_[np.ones(len(X_test)), X_test], y_train, y_test,
     ridge_coefficient, 1 / 0.01)

    """ 1e) """
    phi_train = squad_exponentional_matrix(20, X_train)
    phi_test = squad_exponentional_matrix(20, X_test)
    calc_error_of_train(phi_train, phi_test, y_train, y_test, ridge_coefficient, 1/0.01)
    baysian_regression(phi_train, X_train, y_train, ridge_coefficient, 1/0.01)
    baysian_regression(phi_test, X_test, y_test, ridge_coefficient, 1/0.01)

    """ 1f) """
    beta = [1, 10, 100]
    for b in beta:
        baysian_regression(phi_train, X_train, y_train, 0.01, b)
        baysian_regression(phi_test, X_test, y_test, 0.01, b)
        calc_error_of_train(phi_train, phi_test, y_train, y_test, ridge_coefficient, b)

if __name__ == '__main__':
    main()
```

Listing 17: full code

2 Linear Classification

2.1 2a)

Explain the difference between discriminative and generative models and give an example for each case. Which model category is generally easier to learn and why?

Both predict the conditional probability but both models learn different probabilities.

A Discriminative model models the decision boundary between the classes by assuming some form for P(Class|X) and estimating parameters of P(Class|X) directly from training data. E.g. logistic regression, SVM, neural networks, nearest neighbour, Conditional Random Fields (CRF)s.

A Generative Model models the actual distribution of each class by assuming some forms for P(Class) and P(X|Class), estimating the parameters of P(X|Class) and P(Class) directly from training data and using Bayes rule to calculate P(Class|X). E.g. Naive Bayes, Bayesian networks, Markov random fields, Hidden Markov Models (HMM).

Generally discriminative model is easier to learn because it doesn't involve estimating the prior of the classes and we don't care whether we fit the class-conditional well.

Tutor solution: Discriminative models categorize a sample to a class, but unlike the generative models, can not generate samples from that class. Discriminative models learn how to categorize the data while generative models can learn the distribution of the data, $p(\vec{X} \mid C_i)$. A special case of discriminant models can model the conditional distribuiton $p(C_i \mid \vec{X})$. Examples of generative models are Bayesian classifiers; examples of discriminant models are linear discriminant functions. Generally, discriminative models are easier to learn as they only have to learn how to assign classes to inputs and do not learn the probability distribution of the data

2.2 2b)

Discriminative models map a vector $\boldsymbol{x}$ to one of the K available classes. For example a Bayes classifier that maps $\boldsymbol{x}$ according to a set of decision boundaries.

Given N M-dimensional samples $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, where in this case $N = 137$ and $M = 2$. We know we have $K = 3$ classes, each class has N_i samples: $N_1 = 50$, $N_2 = 43$, and $N_3 = 44$. Stacking the samples into a wide matrix $\boldsymbol{X} \in \mathbb{R}^{M \times N}$ such that each column represents one sample, we want to obtain a transformation of $\boldsymbol{X}$ to $\boldsymbol{Y} = \boldsymbol{W}^T \boldsymbol{X}$ through projecting the samples in $\boldsymbol{X}$ onto a plane (hyperplane of dimension $K - 1 = 2$), where $\boldsymbol{W}$ is the projection matrix used to project $\boldsymbol{X}$ to $\boldsymbol{Y}$,

$$\boldsymbol{W} = [\boldsymbol{w}_1 | \ldots | \boldsymbol{w}_{K-1}] = [\boldsymbol{w}_1 | \boldsymbol{w}_2]$$

The optimal projection matrix $\boldsymbol{W}^*$ is the one whose columns are the eigenvectors corresponding to the largest eigenvalues of the generalized eigenvalue problem

$$\boldsymbol{S}_W^{-1} \boldsymbol{S}_B \boldsymbol{W}^* = \lambda \boldsymbol{W}^*,$$

where **within-class scatter** for K-classes:

$$\boldsymbol{S}_W = \sum_{i=1}^{K} \boldsymbol{S}_i,$$

where

$$\boldsymbol{S}_i = \sum_{\boldsymbol{x} \in K_i} (\boldsymbol{x} - \boldsymbol{\mu}_i) (\boldsymbol{x} - \boldsymbol{\mu}_i)^T$$

the class means

$$\boldsymbol{\mu}_i = \frac{1}{N_i} \sum_{\boldsymbol{x} \in K_i} \boldsymbol{x}$$

and the **between-class scatter**

$$\boldsymbol{S}_B = \sum_{i=1}^{K} N_i (\boldsymbol{\mu}_i - \boldsymbol{\mu}) (\boldsymbol{\mu}_i - \boldsymbol{\mu})^T,$$

where the overall mean

$$\boldsymbol{\mu} = \frac{1}{N} \sum_{\forall \boldsymbol{x}} \boldsymbol{x}.$$

The provided dataset, it classification, and class means are depicted in Figure 4.

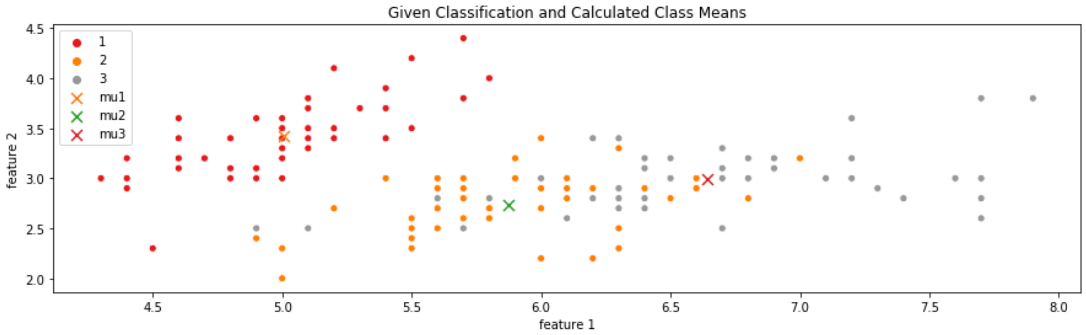


Figure 4: Given dataset and classification

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

def gauss(x,m,var):
    return np.exp(-(x-m)**2/2/var) / np.sqrt(2*np.pi*var)

def multvar_gauss(x,m,S):
    p = m.shape[0]
    return (2*np.pi)**(-p/2) / np.sqrt(np.linalg.det(S)) * np.exp(-0.5*np.matmul(np.transpose(x-m), np.matmul(np.linalg.
     inv(S),(x-m)) ))

x = np.genfromtxt("ldaData.txt")
N = x.shape[0]
# No. of samples per class
N1 = 50
N2 = 43
N3 = 44

# data per class
x1 = x[0:N1,:]
x2 = x[N1:N1+N2,:]
x3 = x[N1+N2:,:]

# The class means
mu1 = np.mean(x1, axis=0).reshape((2,1))
mu2 = np.mean(x2, axis=0).reshape((2,1))
mu3 = np.mean(x3, axis=0).reshape((2,1))

# class vector
c = np.concatenate((np.repeat(1,50), np.repeat(2,43), np.repeat(3,44)))

plt.figure(figsize=(15,4))
sns.scatterplot(x[:,0], x[:,1], hue=c, palette='Set1', legend='full')
plt.scatter(mu1[0,0], mu1[1,0], marker='x', s=80, label='mu1')
plt.scatter(mu2[0,0], mu2[1,0], marker='x', s=80, label='mu2')
plt.scatter(mu3[0,0], mu3[1,0], marker='x', s=80, label='mu3')
plt.title('Given Classification and Calculated Class Means')
plt.xlabel('feature 1')
plt.ylabel('feature 2')
plt.legend()
plt.show()
```

Listing 18: Reading and Plotting of the Given Classification

Using 1-D projection for new classification

The density of the projected samples onto the two eigenvectors are shown in Figure 5.

Using the **first projection vector** (corresponding to the eigenvector with the highest eigenvalue) for maximum separation, find the class of each sample according to Bayes classification,

$$\begin{aligned}\hat{f}(y) &= \arg\max_i p(K_i|y) \\ &= \arg\max_i p(y|K_i)\,\hat{\pi}_i,\end{aligned}$$

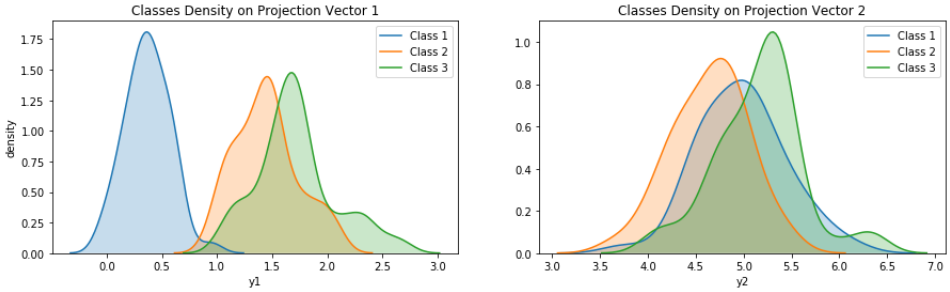


Figure 5: Projection onto the eigenvectors.

where assuming Gaussian, $p\left(y|K_i\right) = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left(-\frac{(y-\mu_i)^2}{2\sigma_i^2}\right)$, where μ_i and σ_i^2 are the mean and variance of the projected samples that belong to the ith class, respectively, and the prior probability for class i, $\pi_i,\ i = 1, 2, 3$, is estimated by the fraction of training samples of class i:

$$\begin{aligned}\hat{\pi}_1 &= \frac{N_1}{N} = \frac{50}{137}\\ \hat{\pi}_2 &= \frac{N_2}{N} = \frac{43}{137}\\ \hat{\pi}_3 &= \frac{N_3}{N} = \frac{44}{137}.\end{aligned}$$

The estimated classes are shown in Figure 6, **number of misclassified points: 30**.

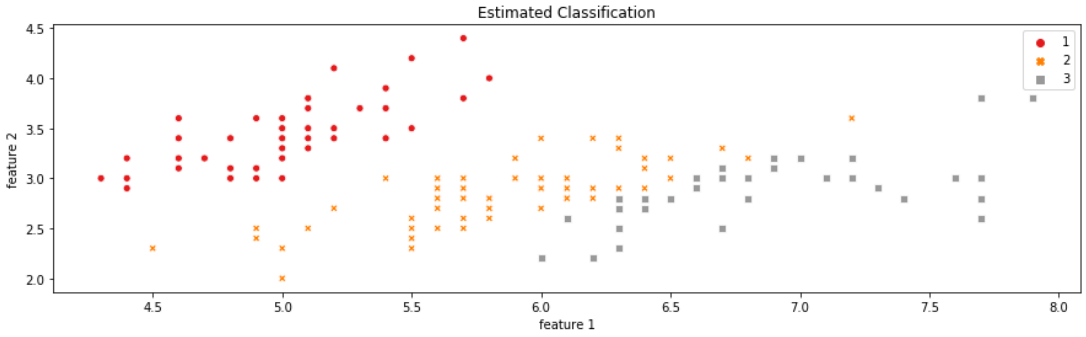


Figure 6: Classification by projection to the first eigenvector.

We observe in the figure that class 1 has clearer separation from the other classes, therefore there is only one misclassified point for this class (1 to another), while classes 2 and 3 are more mixed up, therefore 29 of the 30 errors are misclassifications between these two classes (2 to 3, 3 to 2).

```
# Overall mean
mu = np.mean(x, axis=0).reshape((2,1))

# Class covariance matrices
S1 = np.cov(np.transpose(x1))
S2 = np.cov(np.transpose(x2))
S3 = np.cov(np.transpose(x3))

# Within-class scatter matrix
S_W = S1+S2+S3

# Between-class scatter matrix
S_B1 = N1 * np.matmul( (mu1-mu), np.transpose(mu1-mu) )
S_B2 = N2 * np.matmul( (mu2-mu), np.transpose(mu2-mu) )
S_B3 = N3 * np.matmul( (mu3-mu), np.transpose(mu3-mu) )
S_B = S_B1 + S_B2 + S_B3

ISW_SB = np.dot(np.linalg.inv(S_W), S_B)
# The projection vectors are the eigenvectors
# eigvals are already sorted
```

```
eigvals, eigvecs = np.linalg.eig(ISW_SB)

###################### USING 1-D PROJECTION ######################
# 3 classes => 2 projection vectors
w1 = eigvecs[:,0].reshape(2,1)
w2 = eigvecs[:,1].reshape(2,1)

# Projection onto the first projection vector
y_w1 = np.dot(x,w1).reshape(N)
# Projection onto the second projection vector
y_w2 = np.dot(x,w2).reshape(N)

plt.figure(figsize=(15,4))
plt.subplot(1,2,1)
sns.kdeplot(y_w1[0:N1], shade=True, label='Class 1')
sns.kdeplot(y_w1[N1:N1+N2], shade=True, label='Class 2')
sns.kdeplot(y_w1[N1+N2:], shade=True, label='Class 3')
plt.legend()
plt.title("Classes Density on Projection Vector 1")
plt.xlabel('y1')
plt.ylabel('density')

plt.subplot(1,2,2)
sns.kdeplot(y_w2[0:N1], shade=True, label='Class 1')
sns.kdeplot(y_w2[N1:N1+N2], shade=True, label='Class 2')
sns.kdeplot(y_w2[N1+N2:], shade=True, label='Class 3')
plt.legend()
plt.title("Classes Density on Projection Vector 2")
plt.xlabel('y2')
plt.show()

# Priors
pi1 = 1/N1
pi2 = 1/N2
pi3 = 1/N3

y1 = y_w1[0:N1]
y2 = y_w1[N1:N1+N2]
y3 = y_w1[N1+N2:]
# The new  class means
y_mu1 = np.mean(y1)
y_mu2 = np.mean(y2)
y_mu3 = np.mean(y3)
# The new  class variance
y_var1 = np.var(y1)
y_var2 = np.var(y2)
y_var3 = np.var(y3)

# Classifier
cnew = np.zeros(N, dtype='int8')
for n in range(N):
    Post1 = gauss(y_w1[n], y_mu1, y_var1) * pi1
    Post2 = gauss(y_w1[n], y_mu2, y_var2) * pi2
    Post3 = gauss(y_w1[n], y_mu3, y_var3) * pi3
    cnew[n] = np.argmax((Post1, Post2, Post3)) + 1

# Plot new classification result
plt.figure(figsize=(15,4))
sns.scatterplot(x[:,0], x[:,1], hue=cnew, style=cnew, palette='Set1', legend='full')
plt.xlabel('feature 1')
plt.ylabel('feature 2')
plt.title('Estimated Classification')
plt.legend()
plt.show()

print('Misclassified points: ' + str(np.sum(c!=cnew)))

wrongidx = np.argwhere(c!=cnew)
for i in range(len(wrongidx)):
    print('pt (' + str(x[wrongidx[i],0]) + ','+ str(x[wrongidx[i],1]) +
                '), given: ' + str(c[wrongidx[i]]) +
          ', classified as: ' + str(cnew[wrongidx[i]]))
```

Listing 19: Finding optimal projection vectors and new classification using 1D projection onto the first eigenvector.

Using 2-D projection for new classification

Using both eigenvectors $\boldsymbol{W} = [\boldsymbol{w}_1|\boldsymbol{w}_2]$, the projected samples are $\boldsymbol{Y} = \boldsymbol{W}^T\boldsymbol{X}$. The new classes are now determined by

$$\hat{f}(\boldsymbol{y}) = \arg\max_i p(\boldsymbol{y}|K_i)\,\hat{\pi}_i,$$

where $\boldsymbol{y}$ is each sample in $\boldsymbol{Y}$ and assuming multivariate Gaussian, $p(\boldsymbol{y}|K_i) \sim \mathcal{N}_p(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$ with $\boldsymbol{y}, \boldsymbol{\mu}_i \in \mathbb{R}^M$, $\boldsymbol{\Sigma}_i \in \mathbb{R}^{M\times M}$:

$$p(\boldsymbol{y}|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) = (2\pi)^{-\frac{M}{2}}\,|\boldsymbol{\Sigma}|^{-\frac{1}{2}}\exp\left[-\frac{1}{2}(\boldsymbol{y}-\boldsymbol{\mu}_i)^T\boldsymbol{\Sigma}_i^{-1}(\boldsymbol{y}-\boldsymbol{\mu}_i)\right],$$

with $\boldsymbol{\mu}_i$ and $\boldsymbol{\Sigma}_i$ are the class mean and covariance, estimated from the projected samples.

The estimated classes are shown in Figure 7, **number of misclassified points: 19**.

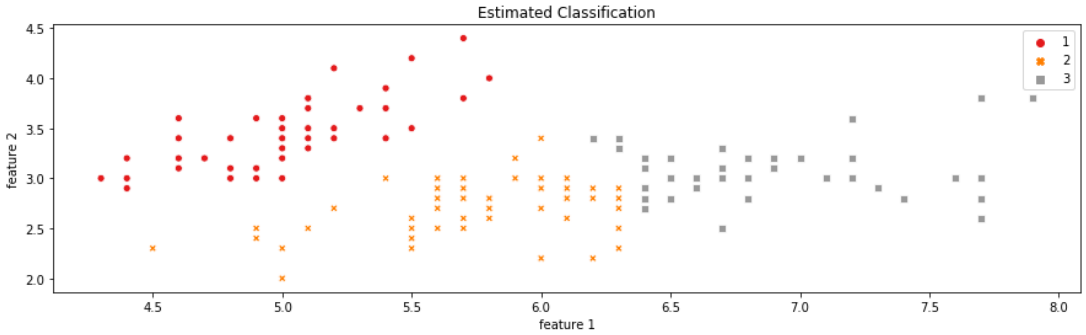


Figure 7: Classification by projection to both eigenvectors.

```
###################### USING 2-D PROJECTION ######################
W = eigvecs # projection matrix
Y = np.dot(x,W)
Y1 = Y[0:N1,:]
Y2 = Y[N1:N1+N2,:]
Y3 = Y[N1+N2:,:]
# The new  class means
Y_mu1 = np.mean(Y1, axis=0)
Y_mu2 = np.mean(Y2, axis=0)
Y_mu3 = np.mean(Y3, axis=0)
# The new  class covariance
YS1 = np.cov(Y1.T)
YS2 = np.cov(Y2.T)
YS3 = np.cov(Y3.T)

cnew = np.zeros(N, dtype='int8')
for n in range(N):
    Post1 = multvar_gauss(Y[n,:], Y_mu1, YS1) * pi1
    Post2 = multvar_gauss(Y[n,:], Y_mu2, YS2) * pi2
    Post3 = multvar_gauss(Y[n,:], Y_mu3, YS3) * pi3
    cnew[n] = np.argmax((Post1, Post2, Post3)) + 1

print('Misclassified points: ' + str(np.sum(c!=cnew)))

# Plot new classification result
plt.figure(figsize=(15,4))
sns.scatterplot(x[:,0], x[:,1], hue=cnew, style=cnew, palette='Set1', legend='full')
plt.xlabel('feature 1')
plt.ylabel('feature 2')
plt.title('Estimated Classification')
plt.legend()
plt.show()
```

Listing 20: New classification using 2-D projection onto both eigenvectors.

*Note that while in the text the samples are collected in a wide matrix, in the code tall matrices are used.

3 Principal Component Analysis

Given a collection of data points $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)}$, $\boldsymbol{x}^{(n)} \in \mathbb{R}^M$, perform a low-dimensional representation

$$\boldsymbol{x}^{(n)} = \boldsymbol{B}\boldsymbol{a}^{(n)} + \boldsymbol{c} + \boldsymbol{v}^{(n)}, \ n = 1, \ldots, N,$$

where

- $\boldsymbol{B} \in \mathbb{R}^{M \times D}$ is a basis matrix, $D < \min\{M, N\}$
- $\boldsymbol{a}^{(n)} \in \mathbb{R}^D$ is the coefficient for $\boldsymbol{x}^{(n)}$,
- $\boldsymbol{c} \in \mathbb{R}^M$ is the base or mean,
- $\boldsymbol{v}^{(n)} \in \mathbb{R}^M$ is the noise/ modeling error.

PCA:

- Choose $\boldsymbol{c} = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{x}^{(n)}$
- Let $\boldsymbol{x}'^{(n)} = \boldsymbol{x}^{(n)} - \boldsymbol{c}$, collate the data into matrix $\boldsymbol{X}' \in \mathbb{R}^{N \times M}$, collate $\boldsymbol{a}^{(n)}$ into $\boldsymbol{A} \in \mathbb{R}^{N \times D}$, and solve

  $$\min_{\boldsymbol{A},\boldsymbol{B}} \left\| \boldsymbol{X}' - \boldsymbol{A}\boldsymbol{B}^\top \right\|_F^2,$$

  where $\|\boldsymbol{Y}\|_F$ is the Frobenius norm $\|\boldsymbol{Y}\|_F = \sqrt{\sum_{i,j} |y_{ij}|^2} = \sqrt{\mathrm{Tr}\left(\boldsymbol{Y}^T\boldsymbol{Y}\right)}$.

In this question, $M = 4$ for 4 features (sepal length, sepal width, petal length, and petal width) and $N = 150$ observations.

We approached this minimization problem by using the projection of the (normalized) data onto the eigenvectors of the data's covariance matrix. The covariance matrix is used because it captures both the spread and the orientation. It therefore can be represented by a vector (eigenvector) that points into the direction of the spread of the data and by a magnitude (eigenvalue) that represents the spread in this direction.

Since we want to look for the vector that points into the direction of the largest variance, we first choose the largest eigenvalue and moving towards the smaller ones until we arrive to the amount of variance explained that we desire.

3.1 3a)

Normalizing is important, because in many cases (like this one), we want to minimize the error based on the distance measure $\left\| \boldsymbol{X}' - \boldsymbol{A}\boldsymbol{B}^\top \right\|_F^2$ that put the same weight (importance) to all of its dimensions. Therefore the features need to be of the same scale otherwise the features that are relatively small will have little to no contributions in shaping the optimal parameters, as the calculations will favor the features with the biggest scale.

To normalize each dimension to have zero mean and unity variance, we standardize it with:

$$x'_j = \frac{x_j - \bar{x}_j}{\mathrm{std}\,(x_j)}, \ j = 1, \ldots M$$

where $\bar{x}_j$ is the sample mean and $\mathrm{std}\,(x_j)$ is the sample standard deviation of dimension j.

```python
import numpy as np

def get_data(name :str) -> np.asarray:
    file = open(name, 'r')
    X = []
    y=[]
    for i in file.readlines():
        line = i.split(',')
        X.append([float(x) for x in line[0:4]])
        y.append(float(line[4]))
    file.close()
    return np.array(X),np.array(y)

## 3a
X,y = get_data('iris.txt')  #read data
X = (X - np.mean(X, axis=0)) / np.std(X, axis=0) #Normalize X
# Sanity check
print(np.mean(X, axis=0))
print(np.std(X, axis=0))
```

Listing 21: Reading and normalizing data.

3.2 3b)

Let $\boldsymbol{C}$ be the covariance matrix of the samples, $\lambda_1, \ldots, \lambda_M$ be the eigenvalues of $\boldsymbol{C}$ and $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_M$ be the corresponding eigenvectors, i.e.

$$\boldsymbol{C}\boldsymbol{u}_j = \lambda_j \boldsymbol{u}_j.$$

Order λ_j's such that $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_M$, the largest eigenvalue λ_1 gives us the maximal variance, and the corresponding eigenvector $\boldsymbol{u}_1$ gives the direction with maximal variance.

Since we want to look for the vector that points into the direction of the largest variance, we first choose the eigenvector that corresponds to the largest eigenvalue and moving towards the smaller ones until we arrive to the amount of variance explained that we desire.

Let the chosen eigenvectors be $\boldsymbol{u}_1, \ldots \boldsymbol{u}_D$ and $\boldsymbol{B} = [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_D] \in \mathbb{R}^{M \times D}$, the projection of the normalized samples onto the selected eigenvectors:

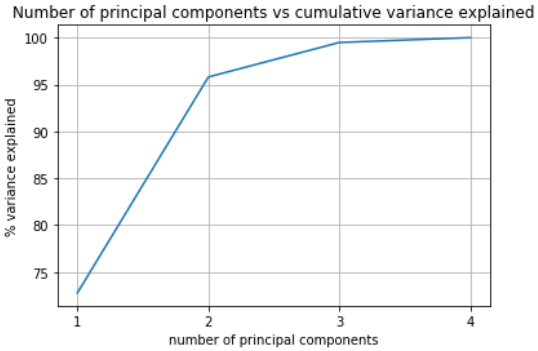
$$\boldsymbol{a}^{(n)} = \boldsymbol{x}'^{(n)} \boldsymbol{B}^\top.$$

The fraction of variance explained by principal components is the ratio between the variance of these principal components and the total variance. Each variance is calculated column-wise on the projected data onto the eigenvectors.

The cumulative variance explained is

$$\text{cve}\,(D) = \frac{\sum_{j=1}^{D} \text{Var}\,(\boldsymbol{a}_j)}{\sum_{j=1}^{M} \text{Var}\,(\boldsymbol{a}_j)} = \frac{\sum_{j=1}^{D} \lambda_j}{\sum_{j=1}^{M} \lambda_j},$$

with $\boldsymbol{a}_j$ the jth column of $\boldsymbol{A} = \boldsymbol{X}'\boldsymbol{B}_M^\top$ with $\boldsymbol{B}_M = [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_M]$ and $\boldsymbol{X}' \in \mathbb{R}^{N \times M}$ the collated samples.



As we can see from the image, slightly above 95% variations in the data are explained with as few as **2 principal components**.

```
############################################################################ 3b
import matplotlib.pyplot as plt

M = X.shape[1] # No. of features
Sigma = np.cov(np.transpose(X)) # Covariance matrix
eigvals, eigvecs = np.linalg.eig(Sigma) # eigvals are already sorted

# total variance
var_total = np.sum( np.var(np.dot(X,eigvecs), axis=0) )
# cumulative variance explained
cve = np.zeros(M)
for D in np.arange(1,M+1):
    B = eigvecs[:,0:D] # matrix of chosen eigenvectors
    A = np.dot(X,B) # projection of the data X to it
    var_D = np.sum( np.var(A, axis=0) ) # explained variance
    cve[D-1] = var_D / var_total * 100 # cumul.percent variance explained

# Plotting
plt.plot(np.arange(1,M+1),cve)
plt.title('Number of principal components vs cumulative variance explained')
plt.xlabel('number of principal components')
plt.xticks(np.arange(1,M+1))
plt.grid()
plt.ylabel('% variance explained')
plt.show()
```

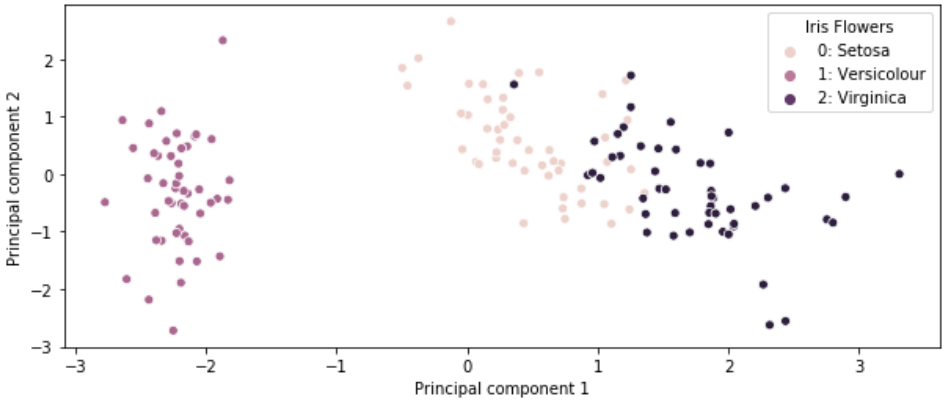
Listing 22: Finding cumulative variance explained for all possible number of principal components.

3.3 3c)

Using $D = 2$ as found from the previous question, the projected samples are

$$\boldsymbol{A} = \boldsymbol{X}' \left([\boldsymbol{u}_1, \ldots, \boldsymbol{u}_2] \right),$$

with $\boldsymbol{A} \in \mathbb{R}^{N \times 2}$.



The figure shows that with half the original dimension, we still have pretty good separability since the two principal components capture more than 95% of variations.

```
############################################################################ 3c
import seaborn as sns

# p=2, eigvecs calculated in 3b
B = eigvecs[:,0:2]
X_transformed = np.dot(X,B) # projection of the data X
# Plotting
plt.figure(figsize=(10,4))
ax = sns.scatterplot(X_transformed[:,0], X_transformed[:,1], hue=y, legend="full")
leg_handles = ax.get_legend_handles_labels()[0]
ax.legend(leg_handles, ['0: Setosa', '1: Versicolour', '2: Virginica'], title='Iris Flowers')
plt.xlabel('Principal component 1')
plt.ylabel('Principal component 2')
plt.show()
```

Listing 23: Transformation to lower dimension with 2 principal components.

3.4 3d)

Let the (unnormalized) back-transformed samples be $\tilde{\boldsymbol{X}}$,

$$\tilde{\boldsymbol{X}} = \left(\boldsymbol{A}\boldsymbol{B}^\top \right) \odot \boldsymbol{\sigma}_X \oplus \boldsymbol{\mu}_X,$$

where $\boldsymbol{B} = [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_D]$, and $\boldsymbol{\mu}_X, \boldsymbol{\sigma}_X \in \mathbb{R}^{N \times D}$ contain the feature-wise mean and standard deviation of the original dataset, respectively, repeated row-wise. The operators $\odot$ and $\oplus$ indicate element-wise multiplication and addition.

The NRMSE for the jth dimension is then [1]

$$\text{NRMSE}_j = \frac{\text{RMSE}_j}{x_{j,\max} - x_{j,\min}} = \sqrt{\frac{1}{N} \sum_{n=1} \left(x_j^{(n)} - \tilde{x}_j^{(n)} \right)^2} \Big/ \left(x_{j,\max} - x_{j,\min} \right),$$

where $\tilde{x}_j^{(n)}$ the nth sample (row) of jth dimension (column) of $\tilde{\boldsymbol{X}}$ for each $j = 1, 2, 3, 4$, and $x_{j,\max} - x_{j,\min}$ is the range of the (unnormalized) samples in jth dimension.

No.of components	x_1	x_2	x_3	x_4
1	0.1040	0.1609	0.0384	0.0831
2	0.0640	0.0170	0.0379	0.0808
3	0.0086	0.0032	0.0343	0.0238
4	0	0	0	0

[1] https://en.wikipedia.org/wiki/Root-mean-square_deviation

```
########################################################################## 3d
def nrmse(x, y): #normalized root mean square error between vectors X and Y
    return np.sqrt(np.mean((x-y)**2)) / np.ptp(x)

X_ori,y_ori = get_data('iris.txt')  #reread unnormalized data
means = np.mean(X_ori, axis=0)
stdevs = np.std(X_ori, axis=0)
# eigenvecs calculated in 3b
for D in np.arange(1,M+1):
    B = eigvecs[:,0:D]
    X_transformed = np.dot(X,B) # X is the normalized data
    X_backtransformed = np.dot(X_transformed, B.T) * stdevs + means #unnormalize
    print("Number of PC: " + str(D))
    for j in np.arange(M):
        print( 'x' + str(j+1) + ': ' + str(nrmse(X_ori[:,j], X_backtransformed[:,j])) )
```

Listing 24: NMRSE calculation

3.5 3e)

Let $\boldsymbol{W}$ be the whitening matrix that satisfies $\boldsymbol{W}^\top \boldsymbol{W} = \boldsymbol{\Sigma}^{-1}$ where $\boldsymbol{\Sigma}$ is the covariance of the design matrix $\boldsymbol{X}$ that contains the samples with mean 0. This results in $\boldsymbol{Y} = \boldsymbol{W}\boldsymbol{X}$ whose covariance matrix is an identity matrix.

1. Explain the difference between PCA and ZCA whitening.
   Mahalanobis or ZCA whitening: $\boldsymbol{W} = \boldsymbol{\Sigma}^{-1/2}$. It uses the sample covariance directly [2].
   PCA whitening: Divide each dimension of the projected samples with the square root of the respective eigenvalue, i.e. $[\boldsymbol{A}_{\text{PCAwhite}}]_j = \frac{[\boldsymbol{A}]_j}{\sqrt{\lambda_j}},\ j = 1, \ldots, D$ with $D \leq M$, where $\boldsymbol{A} = \boldsymbol{X}'\boldsymbol{B} = \boldsymbol{X}'\,([\boldsymbol{u}_1, \ldots, \boldsymbol{u}_D])$.
   $\boldsymbol{u}_j, \lambda_j$ are the eigenvectors and eigenvalues of the covariance matrix of $\boldsymbol{X}'$, and $\boldsymbol{X}'$ the centered dataset with mean 0 in each dimension. PCA whitening uses the eigenvalues and eigenvectors of the (centered) sample covariance and it involves reducing the dimension at the same time.
   Through singular value decomposition, PCA and ZCA whitenings differ in rotation.

2. State the equation(s) to compute the ZCA whitening parameters, given the data.
   Given data $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)}$ that are collated into $\boldsymbol{X} \in \mathbb{R}^{N \times M}$.
   $\hat{\boldsymbol{W}} = \left[\frac{1}{N-1}\sum_{n=1}^{N}(\boldsymbol{x}_n - \hat{\boldsymbol{\mu}})(\boldsymbol{x}_n - \hat{\boldsymbol{\mu}})^\top\right]^{-1/2}$, where $\hat{\boldsymbol{\mu}}$ is the sample mean.

3. State the equation(s) to whiten a (new) data example x, given the ZCA parameters.
   ZCA-whitened data: $\boldsymbol{Y} = \hat{\boldsymbol{W}}\boldsymbol{X}$

4. Compute and report the ZCA whitening parameters for the unnormalized IRIS data (including numerical values!).
   The whitening parameters $\hat{\boldsymbol{W}}$:

$$\hat{\boldsymbol{W}} = \begin{bmatrix} 2.79802739 & -0.94581698 & -1.22308867 & 0.37350572 \\ -0.94581698 & 3.03704291 & 0.86978754 & -0.53844006 \\ -1.22308867 & 0.86978754 & 1.93387305 & -2.02303252 \\ 0.37350572 & -0.53844006 & -2.02303252 & 4.81572824 \end{bmatrix}$$

```
########################################################################## 3e
from scipy.linalg import sqrtm
X,y = get_data('iris.txt')  #reread data, unnormalized
Sigma = np.cov(np.transpose(X))
e = 1e-5
W = np.linalg.inv(sqrtm(Sigma + e))

Y = np.dot(X,W)
print(np.round(np.cov(Y.T),3))
```

Listing 25: ZAC whitening.

Covariance of **Y** after whitening:

```
[[ 1. -0.  0. -0.]
 [-0.  1.  0. -0.]
 [ 0.  0.  1.  0.]
 [-0. -0.  0.  1.]]
```

[2] https://martin-thoma.com/zca-whitening/

Tutor solution:

- In PCE whitening, hte data trainformation to identity covariance is not unique. Any orthogonal matrix R can be multiplied from the left without changing the identity covariance. To make the transformation unique, in ZCA whitening, R is defined to be the matrix of Eigenvectors.

  Let

$$\mu_x = \frac{1}{n}\sum_{i=1}^{n} x_i$$
$$\Sigma_x = \frac{1}{n}(X-\mu_x)^T(X-\mu_x) = \text{USV (using SVD)}$$
$$\sqrt{S_{ii}} := \sqrt{S_{ii}+\epsilon}$$

- then

$$\mu_{ZCA} = \mu_x$$
$$\Sigma_{ZCA} = U(\sqrt{S})^{-1}U^T$$

  where μ_{ZCA} and Σ_{ZCA} are the ZCA whitening parameters.

  To whiten an input x, we compute:

$$\hat{x} = (x-\mu_{ZCA})\Sigma_{ZCA}^T$$

3.6 3f)

Kernel PCA (KPCA) is extension of PCA using kernel method. It is useful for linearly separating data in higher dimensions, when the original data cannot be linearly separated.

When a linear kernel is used, then KPCA is (regular) PCA. **Limitations**: If the data is linearly separable, KPCA can do no better than PCA (maybe even worse due to overfitting). PCA generally is computationally cheaper than KPCA.

Consider mapping data points $\boldsymbol{x}$ to $f(\boldsymbol{x})$ in a higher dimensional feature space. Implementation steps (assuming the samples have zero mean/centered)[3]:

- Choose a kernel $k(\boldsymbol{x}_m, \boldsymbol{x}_n)$, e.g.:
  - Gaussian, $k(\boldsymbol{x}_m, \boldsymbol{x}_n) = \exp\left(-\frac{\|\boldsymbol{x}_m-\boldsymbol{x}_n\|^2}{2\sigma^2}\right)$
  - Polynomial $k(\boldsymbol{x}_m, \boldsymbol{x}_n) = (\boldsymbol{x}_m \cdot \boldsymbol{x}_n)^k$
- Calculate

$$\boldsymbol{K} = \begin{bmatrix} \cdots & \cdots & \cdots \\ \vdots & k(\boldsymbol{x}_i, \boldsymbol{x}_j) & \vdots \\ \cdots & \cdots & \cdots \end{bmatrix}_{N\times N}$$

- Find the eigenvalues λ_i and eigenvectors $\boldsymbol{u}_i$ of $\boldsymbol{K}$, $i = 1, \ldots, N$.
- For each $\boldsymbol{x}$, obtain its principal components in the feature space $\sum_{n=1}^{N} \boldsymbol{u}_{i,n} k(\boldsymbol{x}, \boldsymbol{x}_n)$
- Tutor solution:

  KPCA refuormulates the problem of PCA in a higher -dimensional space using a kernel function. Instead of finding the eigenvectors of the covariance of the dataset, it finds the eigenvector of the covariance of a kernel prjection $\hat{K}$ of the dataset. The projection $\hat{K}$ is a matrix of entries $k_{ij} = \mathcal{K}(\vec{x}_i, \vec{y}_j)$, where $\mathcal{K}$ is a kernel. Applying a kernel transformation, KPCA is able to capture non-linear correlation between input variables. However, it can turn to be more computationally expensive if we have more data points than input variables (which is usually the case), as the matrix $\hat{K}$ has n^2 entries, where n is the number of data points

[3] http://fourier.eng.hmc.edu/e161/lectures/kernelPCA/node4.html