Machine Learning

Summer Semester 2019, Homework 3

Prof. Dr. J. Peters, H. Abdulsamad, S. Stark, D.Koert

TECHNISCHE UNIVERSITÄT DARMSTADT

Total points: 67 + 10 bonus
Due date: Friday, 5 Juli 2019 (17:00)
You need to hand in the pdf in moodle **and** a printed version to the postbox from the IAS secretary office (S2 02 | E315)

Group ID : 146
Name, Surname, ID Number **Peter Nickl, 1941346** **Steffen Schïżœfer, 2635897**

Problem 3.1 Linear Regression [28 Points + 5 Bonus]

In this exercise, you will use the dataset `linRegData.txt`, containing 150 points in the format <`input variable, output variable`>. The input is generated by a sinusoid function, while the output is the joint trajectory of a compliant robotic arm. The first 20 data points are the training set and the remainder are the testing set.

a) **Polynomial Features [10 Points]**

Write the equation of the model and fit it with polynomial features. Using the Root Mean Square Error (RMSE) as a metric for the evaluation, select the complexity of the model (up to a 21st degree polynomial) by evaluating its performance on the testing data. Which is the best RMSE you achieve and what is the model complexity? Does it change if we evaluate our model on the training data? Comment your findings and plot the RMSE for each case (use two lines, one for evaluation on training data, one for evaluation on testing data). For the estimation of the optimal parameters use a ridge coefficient of $\lambda = 10^{-6}$.
Using what you think is the best learned model from the previous point, show in a single plot the ground truth (full dataset) and the model prediction over it. Attach snippets of your code showing how you generate polynomial features and how you fit the model.

First we plot the whole ground truth to get a look at our dataset.

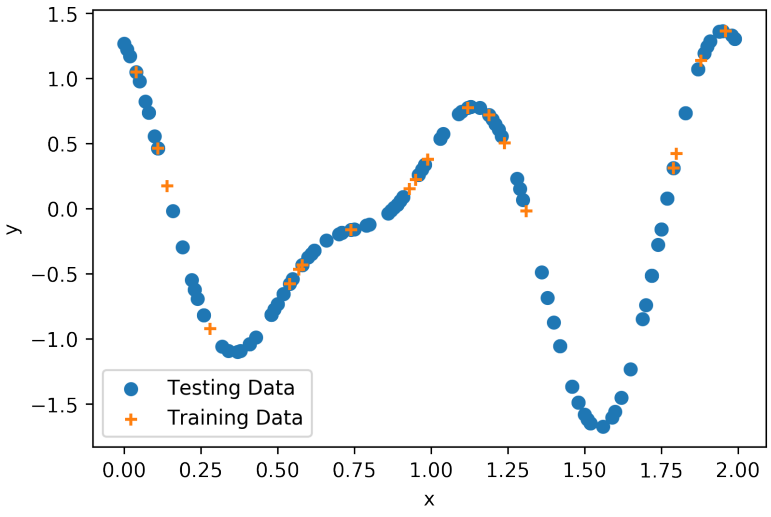

Figure 1: Plotting the whole dataset

Now that we had a look into the data, let's try our polynomial regression with varrying degree. Interesingly we found two different way to calculate our regression based on numericall differences in numpy. We will talk more about this later, but first lets look at the results.

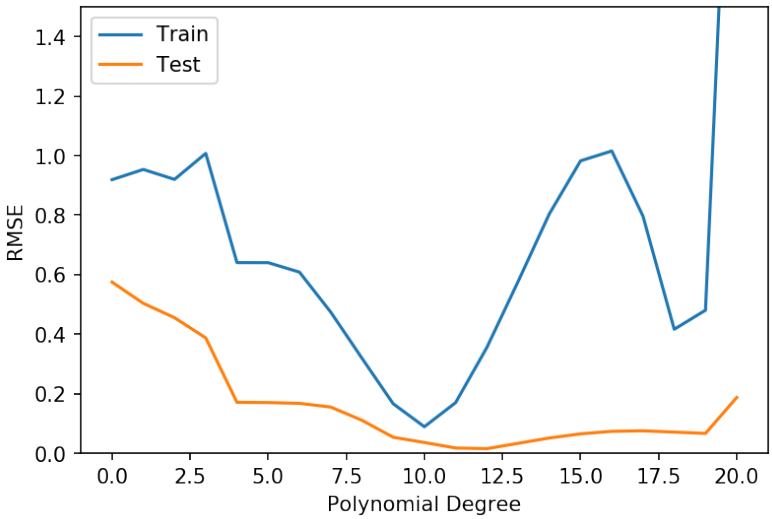

Figure 2: Plotting the error for all different Polynomials using the given ridge coefficient. The minimum lies at degree 10 (RMSE=0.08899).

Lets have a look at the actual predictions given by this model.

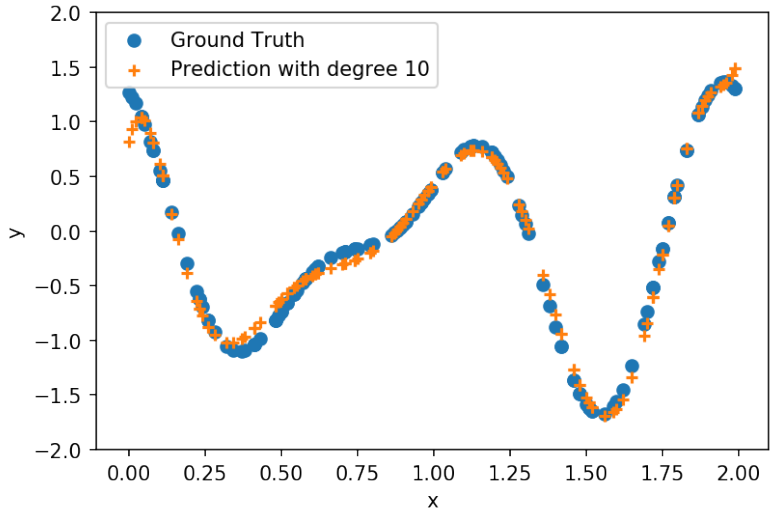

Figure 3: The dataset and the predictions by our model.

However while working on this task, we found a different way to calculate the pseudo inverse. Instead of multiplying all the different matrixes and adding the ridge coefficient we also tried the numpy function *pinv* to calculate the pseudo inverse directly. The method uses singular value decomposition (SVD) to calculate the pseudo inverse and also cuts small values in the diagonal matrix off. This leads to far better results.

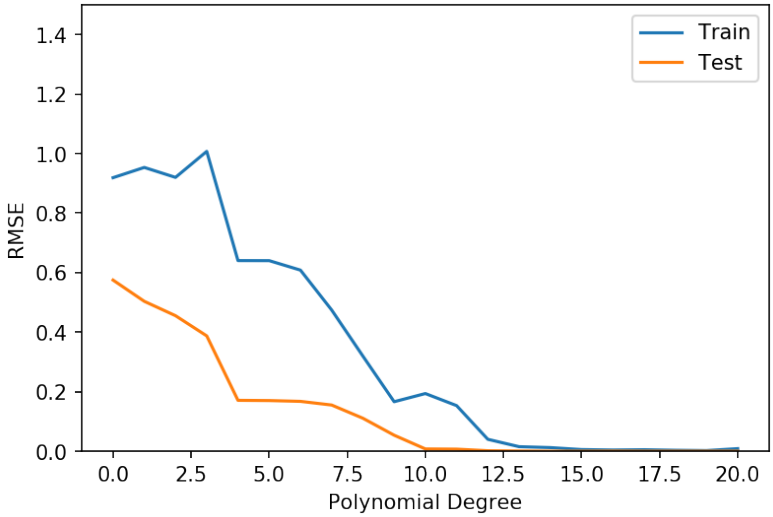

Figure 4: Plotting the error for all different Polynomials using np.linalg.pinv for the pseudo-inverse. The minimum lies at degree 21 (RMSE=0.00491).

This plot 4 only has a single problem. It doesn't tell us when to start overfitting, so there might be the possibility of even better results if we extend the scope of the task to higher polynomials up to degree 25.

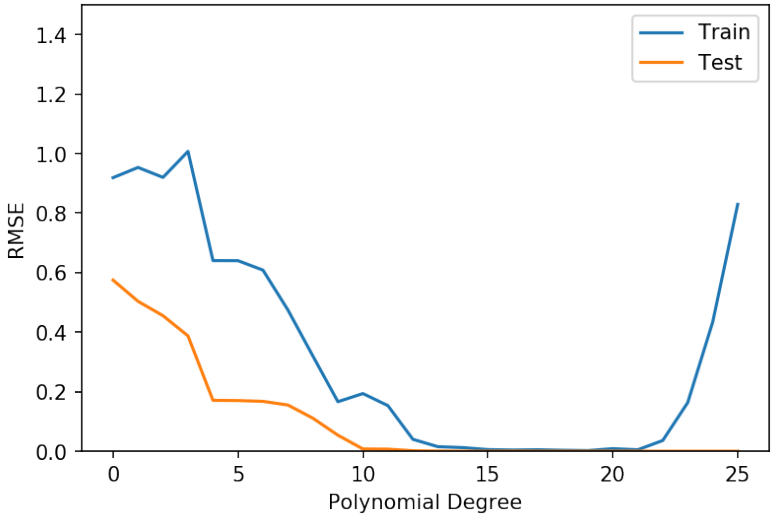

Figure 5: Plotting the error for all different Polynomials using np.linalg.pinv for the pseudo-inverse. The minimum is still at degree 21.

Unfortunately we start overfitting after degree 21. But it was good to check nonetheless. It is also worth noting that the graphs for the version using the ridge coefficient 2 and the graph showing the RMSE for the np.linalg.pinv versions 4 and 5 have the same behavior up to a polynom of 9th degree. After that the SVD version outperforms the ridge version with an RMSE of 0.00491 on the test data for a polynom of degree 21.

For the sake of futher comparision we plot the data and the predictions made by our new model. 6. In comparison the RMSE from the ridge version with 0.08899 is about 18 times bigger, than the RMSE from SVD, however in actual comparision one can argue that the effect of using SVD doesn't perform 18 times better than ridge.

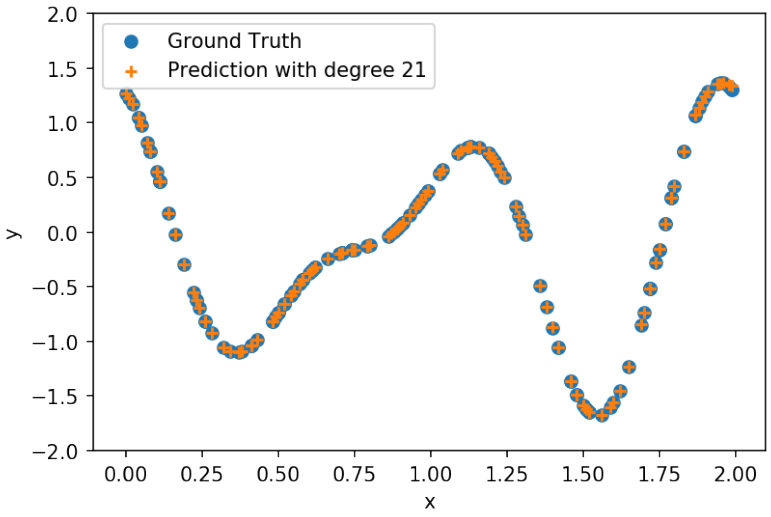

Figure 6: The dataset and the predictions by our model. Even on test data the error is very small

Finally lets have a look at some of the used source code. Depending on which version shall be used either line 25 or 27+28 needs to be commented out.

```
import numpy as np

# Load data
data = np.loadtxt("linRegData.txt")
split_row = 20
train = data[:split_row]
test = data[split_row:]

x_train = np.reshape(train, len(train)*2)[0::2]
y_train = np.reshape(train, len(train)*2)[1::2]
x_test = np.reshape(test, len(test)*2)[0::2]
y_test = np.reshape(test, len(test)*2)[1::2]

# Model Parameter
degree = 21
ridge = 10**(-6)


# Create Design Matrix
Phi = np.zeros(shape=(len(x_train), degree+1))
for i in range(degree+1):
  Phi[:,i] = np.power(x_train, i)

# Calculate left pseudo inverse
#phiPseu = np.linalg.pinv(Phi) # Use this line to let np calculate the pseudo inverse

ridgePhi = np.dot(Phi.T, Phi) + ridge*np.identity(Phi.shape[1])
phiPseu = np.dot(np.linalg.pinv(ridgePhi), Phi.T)


# Model fitting
theta = np.dot(phiPseu, y_train)

# Model prediction
predictions = np.zeros(shape=(len(x_test)))
polynom = np.zeros(shape=(degree+1))
```

```
for i in range(len(x_test)):
  for j in range(degree+1):
    polynom[j] = np.power(x_test[i], j)
  predictions[i] = np.dot(polynom, theta)

# Evaluate RMSE
rmse = np.sqrt(np.mean((predictions-y_test)**2))
print(rmse)
```

b) **Gaussian Features [4 Points]**

Now use Gaussian features. Each feature is a Gaussian distribution were the means are distributed linearly in $x \in [0, 2]$ and the variance is set to $\sigma^2 = 0.02$. The features have to be normalized, i.e., they have to sum to one at every x. Using 10 features generate a plot with the activation of each feature over time (i.e., plot the matrix Φ). Attach a snippet of your code showing how to compute Gaussian features.

Lets have a look at the matrix Phi using 10 Gaussian Features. Since they're all normalized to sum up to one at every point the gaussians at the edge have bigger values.

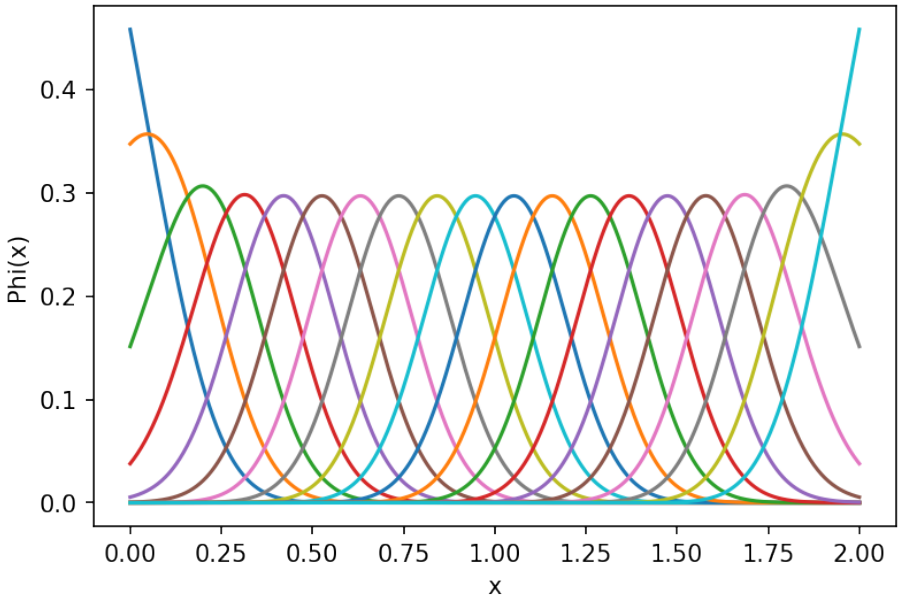

Figure 7: The Design Matrix using 10 Gaussian Features with mean set between 0 and 2.

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


def gaussian(x,mu,h):
    return np.exp(-(x-mu)**2/(2*h**2))/(h*np.sqrt(2*np.pi))


# Load data
data = np.loadtxt("linRegData.txt")
split_row = 30
train = data[:split_row]
test = data[split_row:]

x_train = np.reshape(train, len(train)*2)[0::2]
y_train = np.reshape(train, len(train)*2)[1::2]
x_test = np.reshape(test, len(test)*2)[0::2]
y_test = np.reshape(test, len(test)*2)[1::2]

# Model Parameter
number_features = 20
mean = np.linspace(0,2, number_features)
h = math.sqrt(0.02) # Std
ridge = 10**(-6)

# Create Design Matrix
Phi = np.zeros(shape=(len(x_train), number_features))
sums = np.zeros(len(x_train))
for i in range(number_features):
  Phi[:,i] = gaussian(x_train, mean[i], h) # Use rbf to create PHI

for i in range(len(x_train)):
  sums[i] = np.sum(Phi[i,:]) # After Phi is filled, calculate sums of all rbf
for i in range(len(x_train)):
  Phi[i,:] = np.divide(Phi[i,:], sums[i]) # Normalize so sum = 1

# Calculate left pseudo inverse
v = np.dot(np.transpose(Phi), Phi)
w = np.linalg.inv(v + ridge*np.identity(Phi.shape[1]))
phiPseu = np.dot(w, np.transpose(Phi))

# Model fitting
theta = np.dot(phiPseu, y_train)

# Model prediction
predictions = np.zeros(shape=(len(x_test)))
gaus = np.zeros(shape=(number_features))

for i in range(len(x_test)):
  for j in range(number_features):
    gaus[j] = gaussian(x_test[i], mean[j], h) # Use Gaussian
  gaus = np.divide(gaus, np.sum(gaus)) # Normalize so sum =1
  predictions[i] = np.dot(gaus, theta)

# Evaluate RMSE
rmse = np.sqrt(np.mean((predictions-y_test)**2))
print(rmse)

# Plot Phi
plt.rcParams['figure.dpi']= 150
```

```
plt.xlabel("x")
plt.ylabel("Phi(x)")
axis = np.linspace(0,2, 1000)
Phi = np.zeros(shape=(len(axis), number_features))
sums = np.zeros(len(axis))
for i in range(number_features):
  Phi[:,i] = gaussian(axis, mean[i], h) # Use rbf to create features

for i in range(len(axis)): # Normalize features
  sums[i] = np.sum(Phi[i,:])
  Phi[i,:] = np.divide(Phi[i,:], sums[i])

for i in range(number_features): # Plot Features
  plt.plot(axis, Phi[:,i])
```

c) **Gaussian Features, Continued [4 Points]**

Repeat the process of fitting the model using the Gaussian features from the previous question. Compare the RMSE on the testing data using $15 \ldots 40$ basis functions and plot the RMSE. Which number of basis functions has the best performance and what is the best RMSE? Use a ridge coefficient of $\lambda = 10^{-6}$.

We start by trying out all the given numbers of gaussian features and plot the results. The smalles error (RMSE = 0.01543) is given by using 29 gaussian features 8. In the context of our findings in the last questions its interesting to note, that switching the method of calculating the pseudo inverse using gaussian features does not result in different metrics. Meaning that both methods evaluated in the last questions yield the same result. Hence we will use the ridge version.

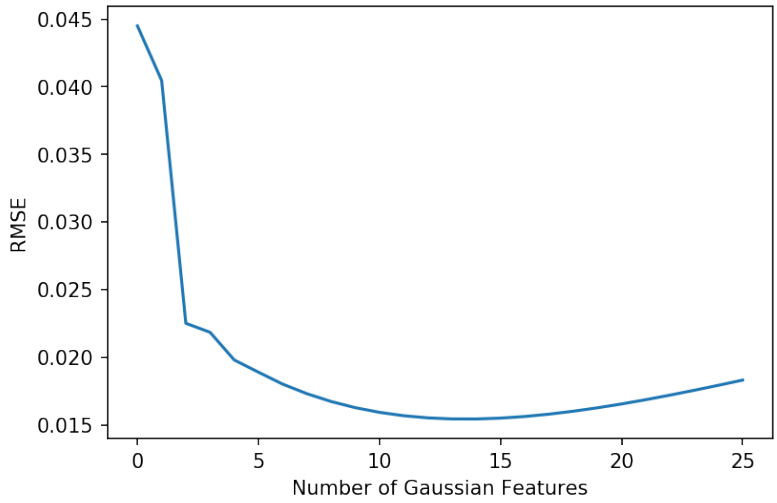

Figure 8: Comparing the number of different gaussian features used, we get that 29 functions perform best.

When looking at the actual predictions made by the model we see that it performs better than the polynomial regression using ridge coefficient, but slightly worse than the polynomial regression using SVD.

When plotting the ground truth and predictions of this model, we see similar performance to polynomial regression with SVD 9.

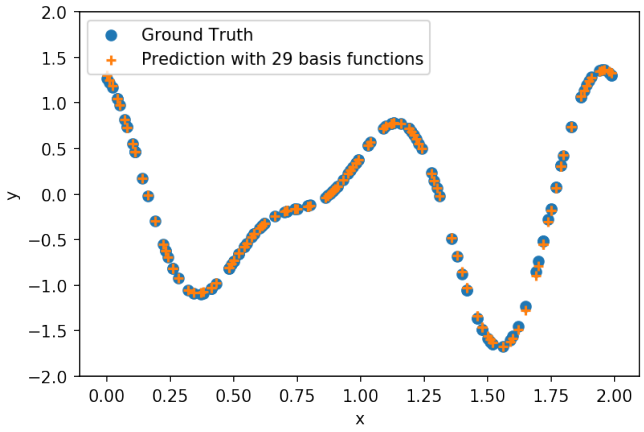

Figure 9: Comparing the actual data with the one produced by the model we get farily good results using gaussian features (RMSE=0.01543).

d) **Bayesian Linear Regression [10 Points]**

Using Bayesian linear regression, plot the mean and the standard deviation of the predictive distribution learned using the first $\{10, 12, 16, 20, 50, 150\}$ data points (one plot per case; plot it in the interval $x \in [0, 2]$). Discuss how the model uncertainty changes with the amount of data points and the problem of overfitting with Bayesian linear regression. Use the best performing polynomial features that you found in 3.1a, a ridge coefficient of $\lambda = 10^{-6}$, and assume Gaussian noise with $\sigma^2 = 0.0025$.

e) **Cross Validation [5 Bonus Points]**

So far, we have split our dataset in two sets: training data and testing data. Cross-validation is a more sophisticated approach for model selection. Discuss it and its variants, pointing out their pro and cons.

Cross validation is used for a number of different techniques in machine learning. In general it means comparing a Trainingset with some kind of Validation or Testset. Using this definition the easiest crossvalidation-method is the holdout method. The holdout method splits the data into three sets: Training-, Validation and Testset. The machine learning Algorithm is trained using the training set and evaluated on the validation set, for tweaking the parameters10. This approach prevents knowledge "leaking" from the Testset into the model. Therefore the metrics used on the testing set will measure the ability of the model to generalize.

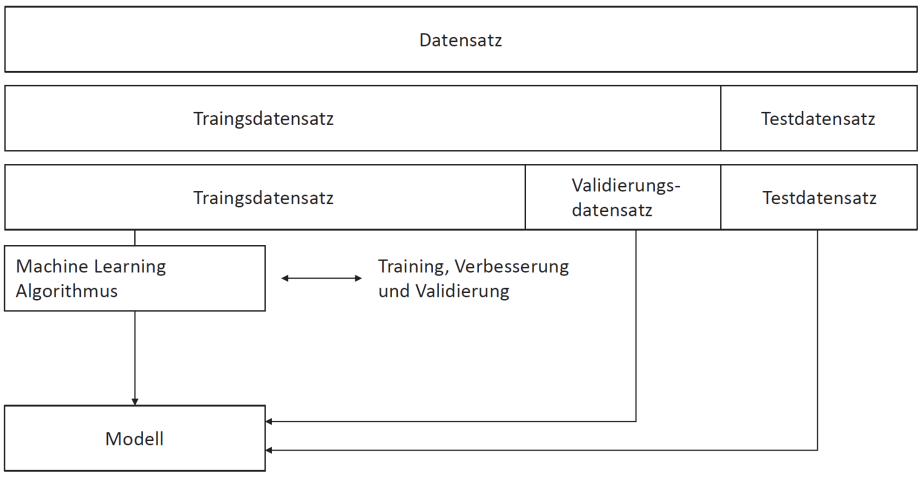

Figure 10: Holdout Cross-Validation schematic

While simple, this approach comes with several limitations. Mainly in reducing the number of training samples for learning the model drastically, but also for introducing a high amount of variance in the evaluation, since the results can depend on a particular random choice of train and validation sets.

To solve this problem k-fold crossvalidation (often just cross-validation, cv) is used. A test set still needs to be hold out for final validation, but we dont need a seperate validation set anymore. Instead we split the training set in k smaller sets. The model is now trained on k-1 sets and validated on the remaining part. This process is now repeated for each of the "k-folds" 10.

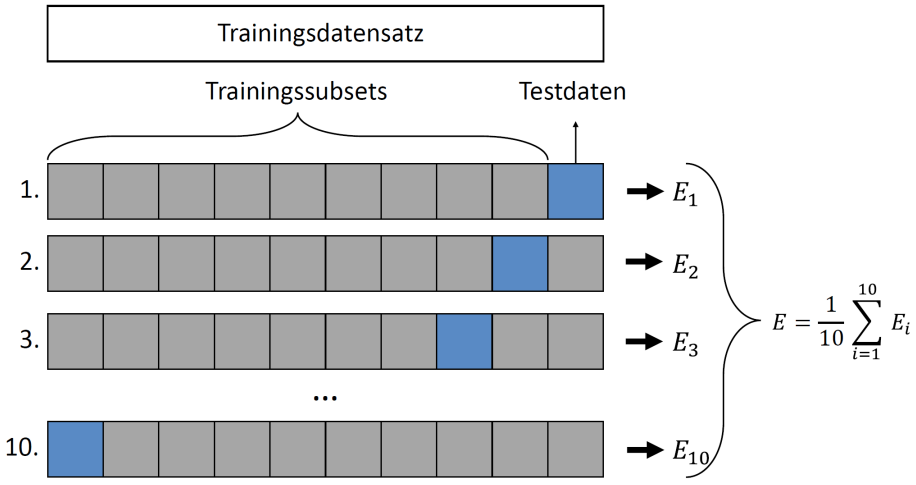

Figure 11: k-Fold cross validation splits the data into k-sets.

The measured perfromance is the average of all individuall runs. This approach can be quite expensive to compute, but uses less data and generally leads to less variance in the outcome.

To futher reduce the variance one can use the stratified k-fold cross-validation method. This method can be used, whenever all data is seen as equally important by the estimator, but may not be in reality. Instead of random shuffling the datasets, the stratified approach tries to create "stratified" folds, meaning that all folds have about the same percentage of samples of each target class, as the complete set. Only works on classification problems.

K-fold cross-validation reaches its limiations when one wants to Hyperparameterize their models. Since no validation set is used, any change of parameters done after seeing the test metrics will lead to overfitting. To bypass this, nested cross-validation is used. In nested cross-validation another loop is added to the process, to be able to hyperparameterize algorithms. Now the inner cross-validation loop is used for training and validating models, while the outer loop is used for changing hyperparameters (for example by using grid search) and to evaluate different model architectures. For example, when using nested cross-validation with neural networks, the inner loop is used to train the network by changing its weights and bias vectors and validate its generalizing performance. Meanwhile the outer loop will be used for changing its hyperparameters e.g. the number of hidden layers and Neurons. The result of the outer loop leads to the best overall model. The excact process is shown in the picture 12.

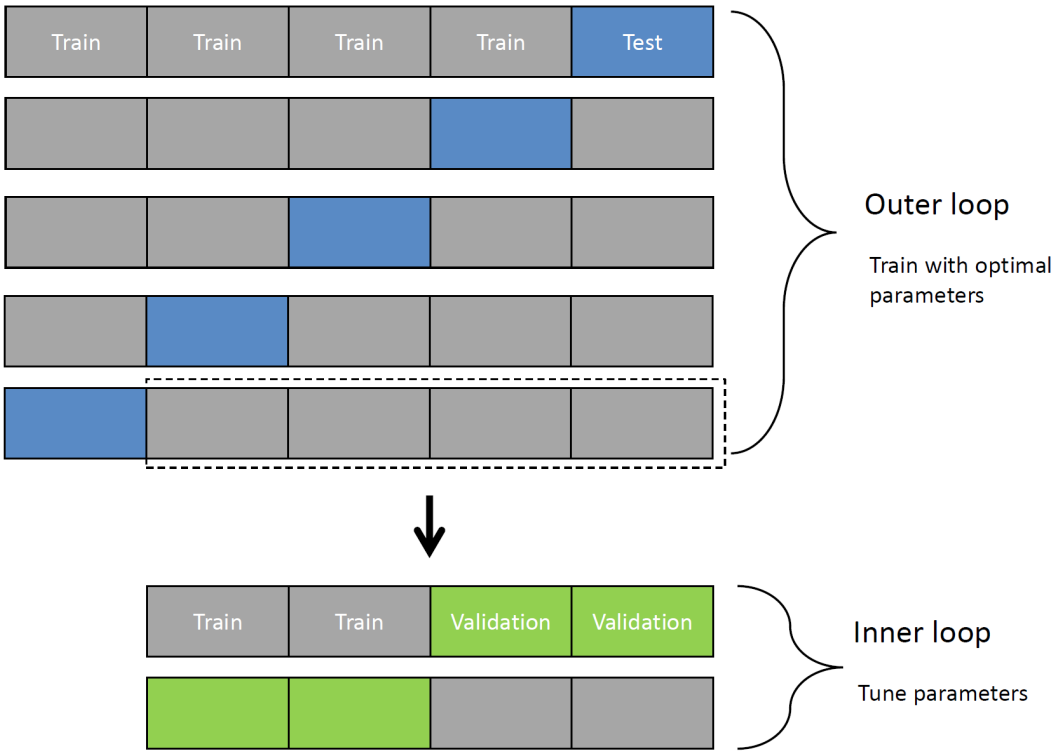

Figure 12: Schematic of nested cross validation. The inner loop is used to train and validate a simple mode, while the outer loop is used to compare different model architecures.

Problem 3.2 Linear Classification [16 Points]

In this exercise, you will use the dataset `ldaData.txt`, containing 137 feature points $\boldsymbol{x}$. The first 50 points belong to class C_1, the second 43 to class C_2, the last 44 to class C_3.

a) **Discriminative and Generative Models [4 Points]**

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?

Generative models *first use the underlying data to deduce properties of the underlying probability distribution in form of the class-doncitional densties $p(x|C_k)$ for each class C_k. Seperatly they infer the prior class probabilities $p(C_k)$. After that, they use Bayes'theorem in the form*

$$\frac{p(x|C_k)\,p(C_k)}{p(x)} \tag{2}$$

to fint the posteriori class probabilities $p(C_k|x)$. Now decision theory is used to determine class membership.

Since in theory we can use these distribution to create new data, this is called a gernative model. Also note that generative models also work when modeling the joint distribution $p(x, C_k)$ and normalizing.

Discriminative models *determine the posterior class probabilities $p(C_k|x)$ directly. After that decision theory is used to each new input to a class.*

Because generative models need the joint distribution $p(x, C_k)$ for calculating the posterior distribution, instead of deriving it directly, they're much harder to train. This is especially true if x is highly dimensional, since then we need a very big dataset to model class-doncitional densities correctly.

b) **Linear Discriminant Analysis [12 Points]**

Use Linear Discriminant Analysis to classify the points in the dataset, i.e., assume Gaussian distributions in each class with equal covariances and use the posterior distributions for assigning classes. Attach two plots with the data points using a different color for each class: one plot with the original dataset, one with the samples classified according to your LDA classifier. Attach a snippet of your code and discuss the results. How many samples are misclassified? (You are allowed to use built-in functions for computing the mean and the covariance.)

First, lets have a look at the data 13:

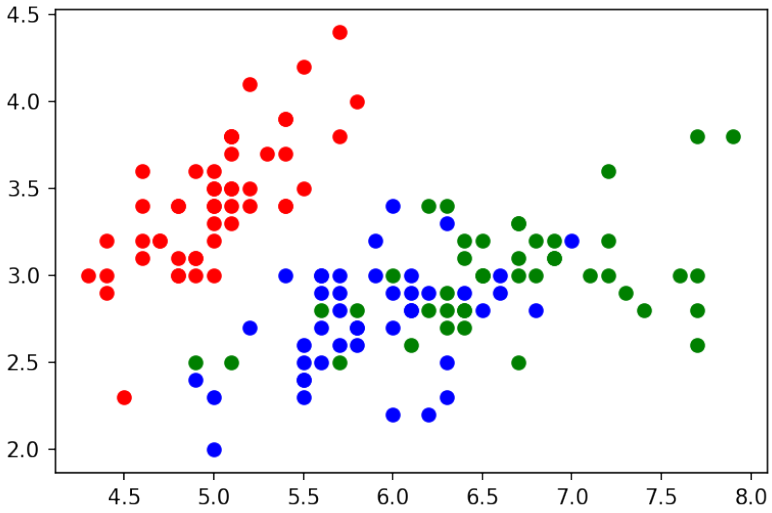

Figure 13: The given data. Three classes of points, represent by the colors: [red, blue, green].

For our classifier we chosed a Maximum Likelihood model with a Gaussian Prior. The classifier yields following result **??**

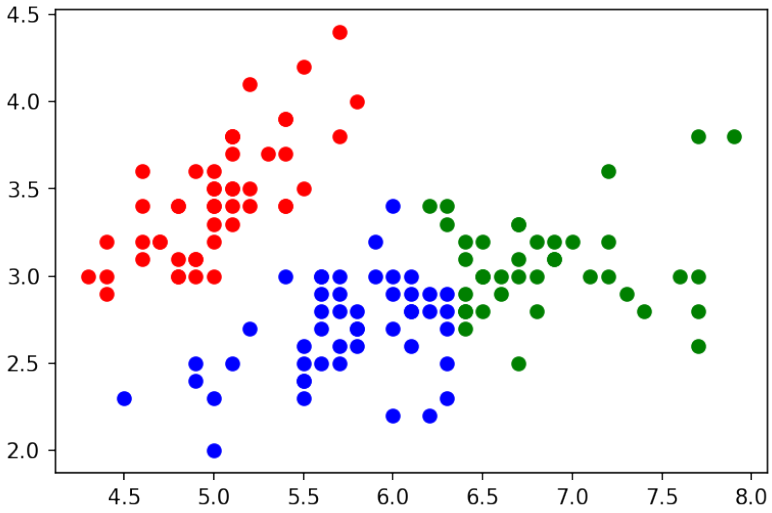

Figure 14: The classified data by our Maximum likelihood. 19 Points are missclassified.

Since there is an overlap between classes two and tree (blue and green), it is impossible to classify all data correctly, without massively overfitting. More features would be needed if we wanted better results, without loosing generalizability. Keeping this in mind, out model performed quite well. The source code used:

```
import numpy as np

# Load Data
data = np.loadtxt("ldaData.txt")
c1 = data[:50] # First 50 values belong to class 1
c2 = data[50:93] # Next 43 to class 2
c3 = data[-44:] # The last 44 to class 3

# Calculate prios
pC1 = len(data) / len(c1)
pC2 = len(data) / len(c2)
pC3 = len(data) / len(c3)

# Calculate mean and variance
mu1 = np.mean(c1, axis=0)
mu1full = np.tile(mu1, (len(c1), 1)) # not in use. Can be used to check for probability if c1 belongs
    into c1
mu2 = np.mean(c2, axis=0)
mu2full = np.tile(mu2, (len(c2), 1))
mu3 = np.mean(c3, axis=0)
mu3full = np.tile(mu3, (len(c3), 1))

s1 = np.std(c1, ddof=1)
cov1 = np.cov(c1, rowvar=False) # Rows = Observations, Column = Variable shapep should be 2x2
s2 = np.std(c2, ddof=1)
cov2 = np.cov(c2, rowvar=False)
s3 = np.std(c3, ddof=1)
cov3 = np.cov(c3, rowvar=False)


def multivariate_gaussian(x, mu, cov):
  i_cov = np.linalg.pinv(cov)
  det_cov = np.linalg.det(cov)

  frac = 1 / np.sqrt((2*np.pi)**len(x) * det_cov)
  exp = np.exp(-1/2 * np.dot(np.dot((x-mu) , i_cov) , (x-mu).T))

  return frac * exp

# Empty lists to store points
listC1 = []
listC2 = []
listC3 = []
for i in range(len(data)):
  posC1 = multivariate_gaussian(data[i], mu1, cov1) * pC1
  posC2 = multivariate_gaussian(data[i], mu2, cov2) * pC2
  posC3 = multivariate_gaussian(data[i], mu3, cov3) * pC3

  if(posC1 > posC2 and posC1 > posC3):
    listC1.append(data[i])
  elif(posC2 > posC1 and posC2 > posC3):
    listC2.append(data[i])
  elif(posC3 > posC1 and posC3 > posC2):
    listC3.append(data[i])
  else:
    print("Couldn't classify point")

predictedC1 = np.array(listC1) # Convert lists to np.arrays
predictedC2 = np.array(listC2)
predictedC3 = np.array(listC3)
```

```
# Plot the results
import matplotlib.pyplot as plt
plt.rcParams['figure.dpi']= 150
plt.scatter(predictedC1[:,0], predictedC1[:,1], c="red")
plt.scatter(predictedC2[:,0], predictedC2[:,1], c="blue")
plt.scatter(predictedC3[:,0], predictedC3[:,1], c="green")
```

Problem 3.3 Principal Component Analysis [23 Points + 5 Bonus]

In this exercise, you will use the dataset `iris.txt`. It contains data from three kind of Iris flowers ('Setosa', 'Versicolour' and 'Virginica') with 4 attributes: sepal length, sepal width, petal length, and petal width. Each row contains a sample while the last attribute is the label (0 means that the sample comes from a 'Setosa' plant, 1 from a 'Versicolour' and 2 from 'Virginica'). (You are allowed to use built-in functions for computing the mean, the covariance, eigenvalues and eigenvectors.)

a) **Data Normalization [3 Points]**

Normalizing the data is a common practice in machine learning. Normalize the provided dataset such that it has zero mean and unit variance per dimension. Why is normalizing important? Attach a snippet of your code.

In machine learning some methods like PCA take the variance of the features as a criterion for dimensionality reduction. If the feature differ in their range, the means and the variances of the features will have a different scale. To achieve better comparability among the features it is common to normalize the data to unit variance and zero mean. The normalization can be achieved by applying the following formula to all datapoints x_i for each feature separately:

$$\frac{x_i - mean_{feature}}{standard\_deviation_{feature}}$$

We calculated the means of the features as follows:

$$means = \begin{bmatrix} 5.84333333 & 3.054 & 3.75866667 & 1.19866667 \end{bmatrix}$$

We calculated the standard deviation of the features as follows:

$$standard\_deviation = \begin{bmatrix} 0.82530129 & 0.43214658 & 1.75852918 & 0.76061262 \end{bmatrix}$$

We apply the normalization to the dataset according to the following code snippet.

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

# set the working directory
os.chdir("C:/Users/pistl/Desktop/SML/homework/homework3/")

# print the current working directory
os.getcwd()

# read in data
def load_data():
    data = np.loadtxt(fname =
        "C:/Users/pistl/Desktop/SML/homework/homework3/dataSets/iris.txt",delimiter=',', skiprows=0)
    return data

# calculate mean and standard_deviation
```

```
def calculate_stats(data):
    mean = np.mean(data[:,:4],axis=0)
    std_deviation = np.std(data[:,:4],axis=0)
    #eigen = np.linalg.eig(data)

    return mean, std_deviation

# normalize data to zero mean and unit variance
def normalize_data(data, mean, std_deviation):
    data_normalized = ( data[:,:4] - mean ) / std_deviation
    return data_normalized

# read in data
data = load_data()

# calculate mean and standard_deviation
stats = calculate_stats(data)
mean = stats[0]
std_deviation = stats[1]

# normalize data to zero mean and unit variance
data_normalized = normalize_data(data,mean,std_deviation)
```

b) **Principal Component Analysis [8 Points]**

Apply PCA on your normalized dataset and generate a plot showing the proportion (percentage) of the cumulative variance explained. How many components do you need in order to explain at least 95% of the dataset variance? Attach a snippet of your code.

For doing the PCA, after normalizing, we have to compute the covariance matrix and subsequently calculate the eigenvectors and eigenvalues of the covariance matrix.
As Figure 15 shows, by using two principal components we can explain 95,8% of the cumulative variance. The four bars are calculated by using the following formula, whereas the eigenvalues are sorted from the biggest eigenvalue to the smallest eigenvalues:

$$\textit{proportion of cumulative variance explained by n biggest eigenvectors} = \frac{\textit{sum of n biggest eigenvalues}}{\textit{sum of all eigenvalues}}$$

For the covariance matrix of the normalized data we obtain:

$$\Sigma = \begin{bmatrix} 1.00671141 & -0.11010327 & 0.87760486 & 0.82344326 \\ -0.11010327 & 1.00671141 & -0.42333835 & -0.358937 \\ 0.87760486 & -0.42333835 & 1.00671141 & 0.96921855 \\ 0.82344326 & -0.358937 & 0.96921855 & 1.00671141 \end{bmatrix}$$

The eigenvalues have the following values:

$$eigenvalues = \begin{bmatrix} 2.93035378 & 0.92740362 & 0.14834223 & 0.02074601 \end{bmatrix}$$

The eigenvectors have the following values. The eigenvectors are sorted according to their respective eigenvalues above. Each column contains one eigenvector.

$$eigenvectors = \begin{bmatrix} 0.52237162 & -0.37231836 & -0.72101681 & 0.26199559 \\ -0.26335492 & -0.92555649 & 0.24203288 & -0.12413481 \\ 0.58125401 & -0.02109478 & 0.14089226 & -0.80115427 \\ 0.56561105 & -0.06541577 & 0.6338014 & 0.52354627 \end{bmatrix}$$

The following code snippet shows the code for doing the PCA and for plotting.

```
# do PCA
def PCA(data_normalized):
    cov = np.cov(data_normalized,rowvar=False)
    eigenvalues, eigenvectors = np.linalg.eig(cov)
    explained_var = eigenvalues / sum(eigenvalues)
    return eigenvalues, eigenvectors, explained_var

eigenvalues = PCA(data_normalized)[0]
eigenvectors = PCA(data_normalized)[1]
explained_var = PCA(data_normalized)[2]

# plot explained cumulative variance over number of principal components
fig, ax = plt.subplots()
explained_var_cumulative = np.cumsum(explained_var)
objects = ('1', '2', '3', '4')
y_pos = np.arange(len(objects))
plt.axhline(y=0.95, xmin=0, xmax=4)
plt.xticks(y_pos, objects)
plt.yticks(np.arange(0, 1.05, step=0.05))
plt.ylabel('cumulative explained variance of dataset')
plt.xlabel('number of principal components')
```

```
plt.title('cumulative explained variance of dataset over number of components')
rects = ax.bar(y_pos, np.around(explained_var_cumulative, decimals=3))
def autolabel(rects):
    for rect in rects:
        height = rect.get_height()
        ax.annotate('{}'.format(height),
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 3), # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom')
autolabel(rects)
plt.savefig('explained_var_cumulative.png',dpi=300)
plt.show()
print(explained_var_cumulative)
```

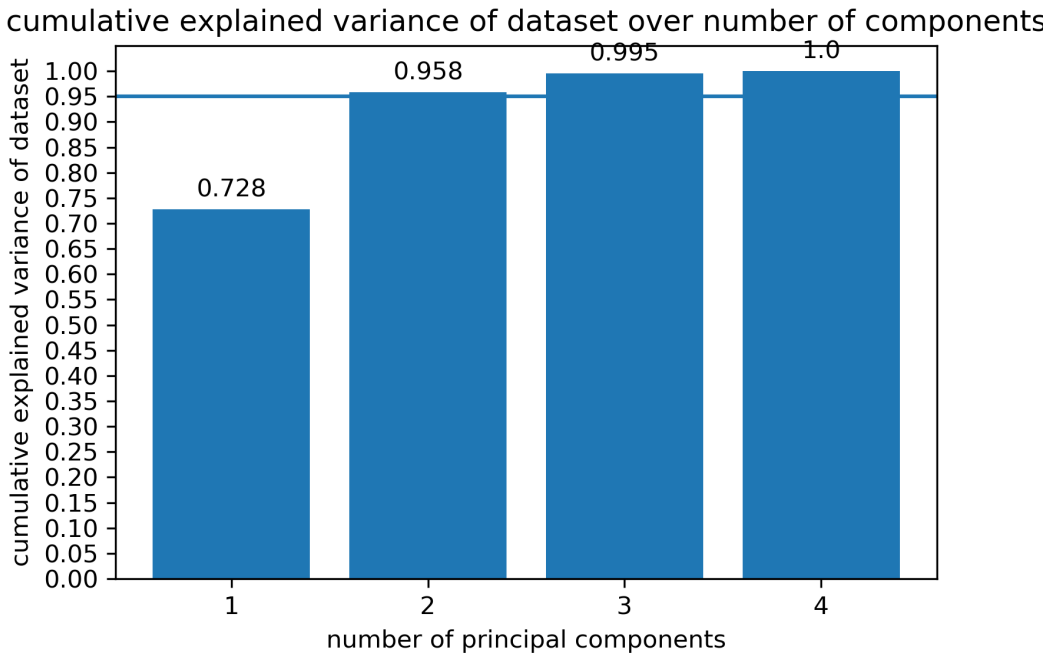

Figure 15: Cumulative explained variance of dataset over number of principal components.

c) **Low Dimensional Space [6 Points]**

Using as many components as needed to explain 95% of the dataset variance, generate a scatter plot of the lower-dimensional projection of the data. Use different colors or symbols for data points from different classes. What do you observe? Attach a snippet of your code.

For projecting the data to the lower dimensional space we use the following formula on the normalized dataset:

$$\mathbf{a}^n = \mathbf{B}^\top \mathbf{x}^n \tag{4}$$

$\mathbf{a}^n$ *is the projected data,* $\mathbf{B}$ *is a matrix containing the eigenvectors - in our case the two eigenvectors corresponding to the two biggest eigenvalues, for obtaining 2-dimensional projection - and* $\mathbf{x}^n$ *is the projected data. The two biggest eigenvalues with corresponding eigenvectors are:*

$$eigenvalues = \begin{bmatrix} 2.93035378 & 0.92740362 \end{bmatrix}$$

The eigenvectors have the following values. The eigenvectors are sorted according to their respective eigenvalues above. Each column contains one eigenvector.

$$eigenvectors = \begin{bmatrix} 0.52237162 & -0.37231836 \\ -0.26335492 & -0.92555649 \\ 0.58125401 & -0.02109478 \\ 0.56561105 & -0.06541577 \end{bmatrix}$$

The 2-dimensional projection leads to the scatter plot in Figure 16.
We can observe that the Versicolour class is clearly separated from the Setosa and Verginica classes in the two-dimensional space. Between the Setoas and the Viriginica class, there is some overlapping, but the class means and variances are still different from each other. Since the two-dimensional projection contains over 95% of the variance of the unprojected dataset, these results can be transferred to the unprojected dataset, too.
Building upon the previously used code, we have used the following code for the two-dimensional projection and the scatter plot.

```
#project data into lower dimensional space
def projection(data_normalized, eigenvectors, num_components):
    eigenvectors = eigenvectors[:,:num_components]
    data_normalized = np.matmul(np.transpose(eigenvectors),np.transpose(data_normalized))
    return data_normalized

#plot lower dimensional space
num_components = 2
projected_data_2comp = projection(data_normalized,eigenvectors,num_components)
x = projected_data_2comp[0,:]
y = projected_data_2comp[1,:]
classes = data[:,4]
colors = ['red','green','blue']
unique = list(set(classes))
plant = None
for i, u in enumerate(unique):
    xi = [x[j] for j in range(len(x)) if classes[j] == u]
    yi = [y[j] for j in range(len(x)) if classes[j] == u]
    if colors[i] == 'red':
        plant = 'Setosa'
    elif colors[i] == 'green':
        plant = 'Versicolour'
    elif colors[i] == 'blue':
        plant = 'Virginica'
    plt.scatter(xi, yi, c=colors[i], label=plant)
```

```
plt.legend()
plt.title("2-dimensional projection of the data")
plt.savefig('PCA_scatter.png',dpi=300)
plt.show()
```

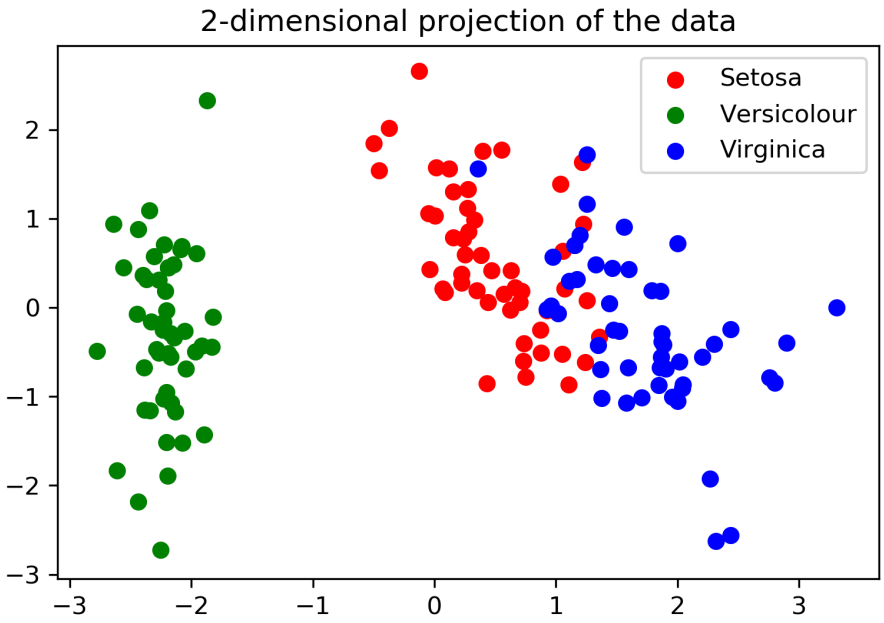

Figure 16: Scatter plot of the 2-dimensional projection of the data.

d) **Projection to the Original Space [6 Points]**

Reconstruct the original dataset by using different number of principal components. Using the normalized root mean square error (NRMSE) as a metric, fill the table below (error per input versus the amount of principal components used).

N. of components	x_1	x_2	x_3	x_4
1				
2				
3				
4				

Attach a snippet of your code. (Remember that in the first step you normalized the data.)

For reconstructing the data, we use the following formula:

$$\tilde{\mathbf{x}}^n = \left(\mathbf{B}\mathbf{a}^n \cdot standard\_deviation_{feature}\right) + mean_{feature}$$

B *contains as many eigenvectors as principal components we use for reconstructing the dataset.* $\mathbf{a}^n$ *is the lower dimensional projection of the data. We also need the standard deviation and mean of each feature, in order to undo the normalization step.*
We then compute the NRMSE for every feature as follows:

$$NRMSE = \frac{RMSE}{x_{max} - x_{min}}$$

x_{max} *and* x_{min} *are the minimal and maximal values of the respective features. The RMSE is calculated as follows:*

$$\text{RMSE} = \sqrt{\frac{\sum_{n=1}^{N} (\tilde{x}_n - x_n)^2}{N}}$$

$\tilde{x}_n$ *are the reconstructed values of the features and* x_n *are the values from the original dataset. By this we can fill in the table as follows:*

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	*7.80185215e-17*	*8.17102022e-17*	*9.95720008e-17*	*6.11515656e-17*

It can be noted, that the NRMSE is extremely small, but not zero, when all four components are used to reconstruct the data. This is due to numerical inaccuray. Building upon the previously show code, this is the code we used for the projection to the original space and the calculation on the NRMSE:

```
#project data back to orgiginal space
def reconstruct_data(data_normalized, eigenvectors, num_components, mean, std_deviation):
    projected_data = projection(data_normalized, eigenvectors, num_components)
    data_reconstructed = np.transpose((np.matmul(eigenvectors[:,:num_components],projected_data)))
    data_reconstructed = ( data_reconstructed *std_deviation ) + mean
    return data_reconstructed

def nrmse(data_reconstructed,data):
    y_max = np.amax(data,axis=0)
    y_min = np.amin(data,axis=0)
    error = data - data_reconstructed
    error_squared = error**2
```

```
    error_squared_sum = np.sum(error_squared,axis=0)
    nrmse = np.sqrt( 1/data.shape[0] * ( error_squared_sum ) ) / ( y_max - y_min )
    return nrmse

num_components = 0
for i in range(4):
    num_components += 1
    data_reconstructed = reconstruct_data(data_normalized, eigenvectors, num_components, mean,
        std_deviation)
    print(nrmse(data_reconstructed,data[:,:4]))
```

e) **Kernel PCA [5 Bonus Points]**

Throughout this class we have seen that PCA is an easy and efficient way to reduce the dimensionality of some data. However, it is able to detect only linear dependences among data points. A more sophisticated extension to PCA, *Kernel PCA*, is able to overcome this limitation. This question asks you to deepen this topic by conducting some research by yourself: explain what Kernel PCA is, how it works and what are its main limitations. Be as concise (but clear) as possible.

As opposed to standard PCA, Kernel PCA is mapping the data into higher dimensional space, where the dimensions d are bigger are equal to the datapoints N. ($d \geq N$)
In this higher dimensional space points can always be linearly separated, as the dimensions are at least as high as the number of data points. Due to this property Kernel regression can overcome the limitation to only detect linear dependencies.
For the transformation into higher dimensional space a arbitrary function Φ is chosen:

$$\Phi(\mathbf{x}_i) \text{ where } \Phi : \mathbb{R}^d \rightarrow \mathbb{R}^N \tag{9}$$

As there is a transformation into higher dimensional space, there is no covariance on which we could perform eigendecomposition as in linear PCA. We can use the Kernel-trick, in order to avoid doing explicit calculations Φ-Space (which is also called feature space). This is done by creating a NxN Kernel, so that:

$$K = k(\mathbf{x}, \mathbf{y}) = (\Phi(\mathbf{x}), \Phi(\mathbf{y})) = \Phi(\mathbf{x})^T \Phi(\mathbf{y}) \tag{10}$$

K can, for example, be the radial basis function Kernel:

$$k(x, y) = e^{\frac{-(||x-y||)^2}{2\sigma^2}} \tag{11}$$

By using the Kernel trick, we never do calculations in the feature space directly. As a result Kernel PCA can not compute the principal components itself, but only directly the projections of out data onto those components, according to the following formula:

$$a^n = K \cdot x^n \tag{12}$$

Apart of the flaw of Kernel PCA, when it comes to computing the principal components itself, Kernel PCA is inefficient for big datasets. The reason for this is, that storing the Kernel gets quite difficult in that case.
Also in practice the choice of a suitable Kernel plays an important role for the performance of Kernel PCA.