Machine Learning

TECHNISCHE UNIVERSITÄT DARMSTADT

Summer Semester 2016, Homework 3 (75 points + 20 bonus)
Prof. Dr. J. Peters, F. Veiga, S. Parisi

Due date: Wednesday, 22 June 2016 (before the lecture)

Problem 3.1 Linear Regression [35 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 $N = 20$ 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 the on 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.

In order to create polynomial features of degree n, the function exprepeat is used. It calculates the matrix X, based on the given vector $x = \begin{pmatrix} x_k \\ \vdots \\ x_1 \\ x_0 \end{pmatrix}$ *as* $X = \begin{pmatrix} x_0^n & x_1^n & \dots & x_k^n \\ \vdots & \ddots & \dots & \vdots \\ x_0 & x_1 & \dots & x_k \\ 1 & 1 & \dots & 1 \end{pmatrix}$

```
def exprepeat(x_in, n):
    X_out = np.ones([x_in.size, 1])

    for i in range(0, degree):
        v = np.array([X_out[:, 0]]).T
        v = np.multiply(v, x_in)
        X_out = np.concatenate((v, X_out), axis=1)

    X_out = X_out.T
    return X_out
```

The polynomial features are then used in polyregress to calculate the coefficients of the polynomial of degree n, incorporating the ridge coefficient $\lambda = 10^{-6}$ *according to the formula in the lecture:* $w = \left(XX^T + \lambda\Sigma\right)^{-1} Xy$.

```
def polyregress(x, y, degree):
    X_p = exprepeat(x, degree)
    w = np.dot(X_p, X_p.transpose())
    w = w + (10**(-6))*np.eye(len(w))
    w = np.dot(np.linalg.inv(w), X_p)
    w = np.array([np.dot(w, y)])
```

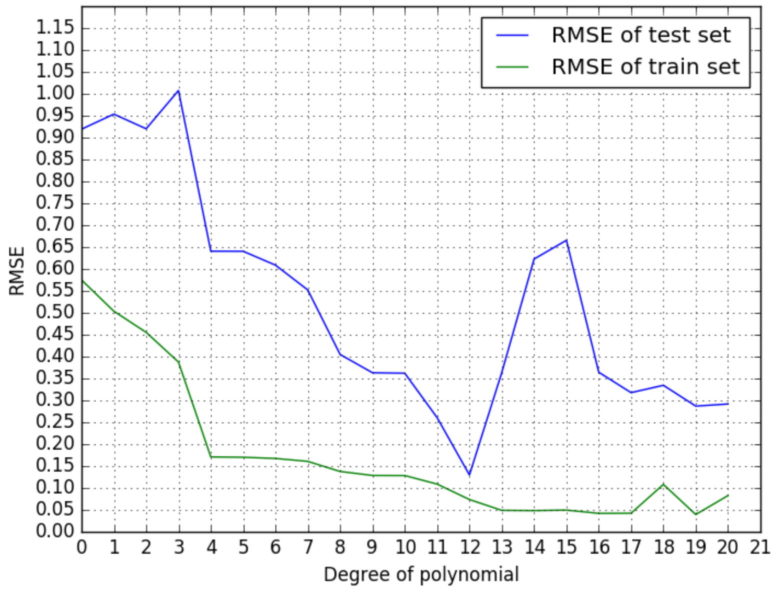
Name, Vorname: ____________________ Matrikelnummer: └┘└┘└┘└┘└┘└┘└┘

```
    return w, X_p
```

Based on these results, the polynomial is first evaluated at all available data points, using $y_{fit} = w^T X$*. The RMSE is than calculated as* $\sqrt{\sum_{n=0}^{N-1} \frac{(y_{fit,n} - y_n)^2}{N}}$*, where N is the number of samples in the dataset. This is done separately for the test and training set.*

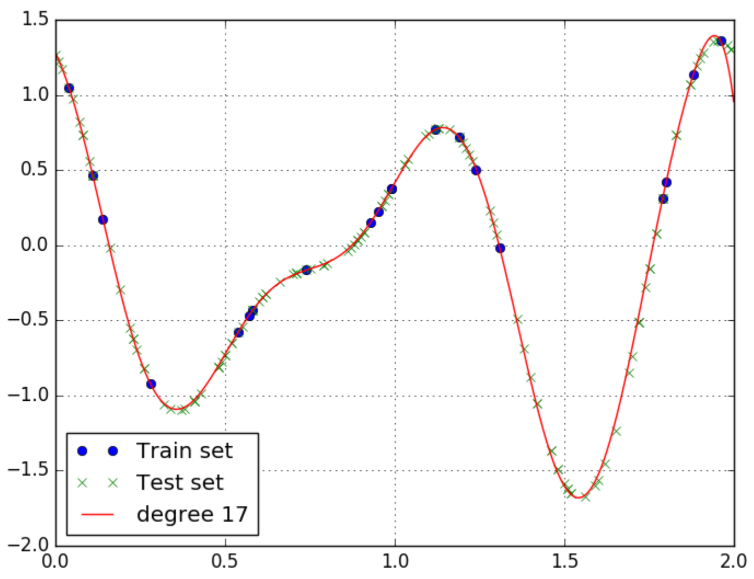
```
def rmse(y, y_fit):
    e = np.subtract(y, y_fit) ** 2
    e = np.sqrt([[np.divide(np.sum(e), e.size)]])
    return e
```

The resulting RMSE for train and test data are plotted in the follwing figure.

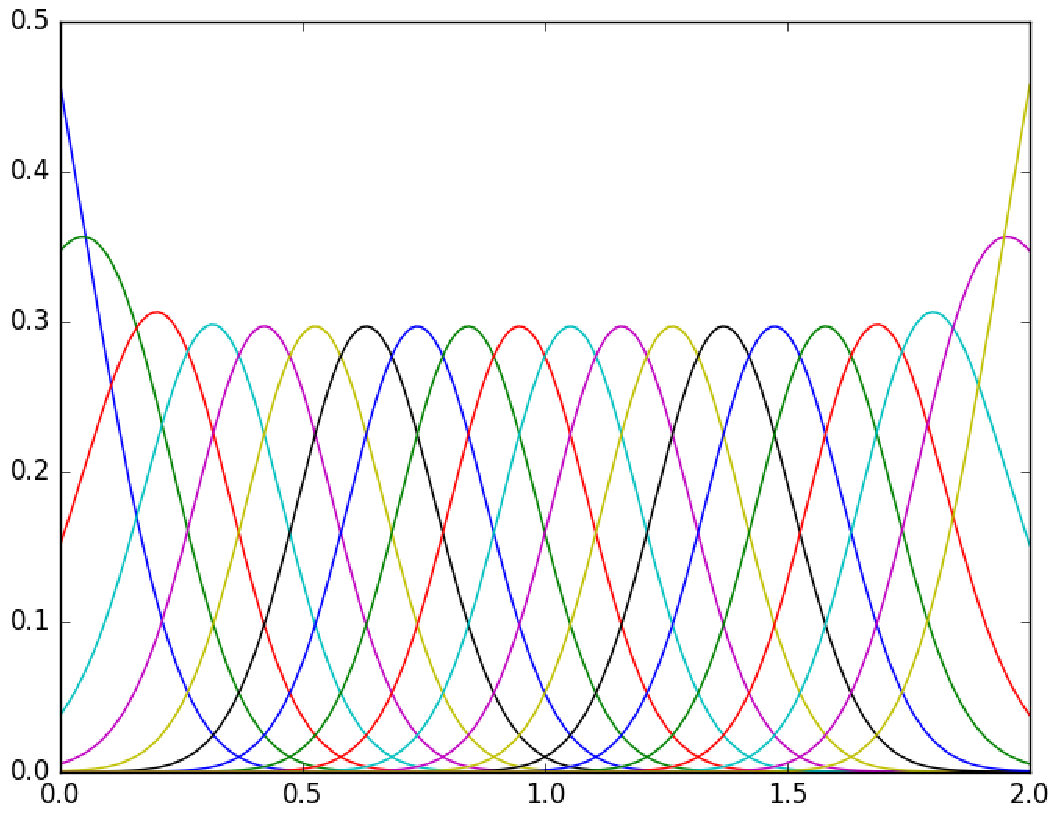


As the plot shows, the RMSE of the training set is always lower than the one of the test set. This is not surprising, since the regression is trying to minimize the error for the training set. The lowest RMSE for the test set is achieved by using a polynomial of degree 17. This polynomial is shown in the following figure.

Name, Vorname: ______________________ Matrikelnummer: ⌴⌴⌴⌴⌴⌴⌴



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 state. Using $N = 20$ 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.



```
# normalizes columns
def normalize(matrix):
        for i in range(0, len(matrix[0])):
                matrix[:, i] = matrix[:, i]/np.sum(matrix[:, i])
        return matrix

# creates phi matrix with gaussians
def get_phi_gaussians(space, x, var):
        matrix = np.zeros((len(x), len(space)))
        for i in range(0, len(matrix[0])):
                matrix[:, i] = x
        for i in range(0, len(matrix)):
                matrix[i] = matrix[i] - space
        matrix = np.exp(-np.power(matrix, 2)/(2*var))
        return matrix.T
```

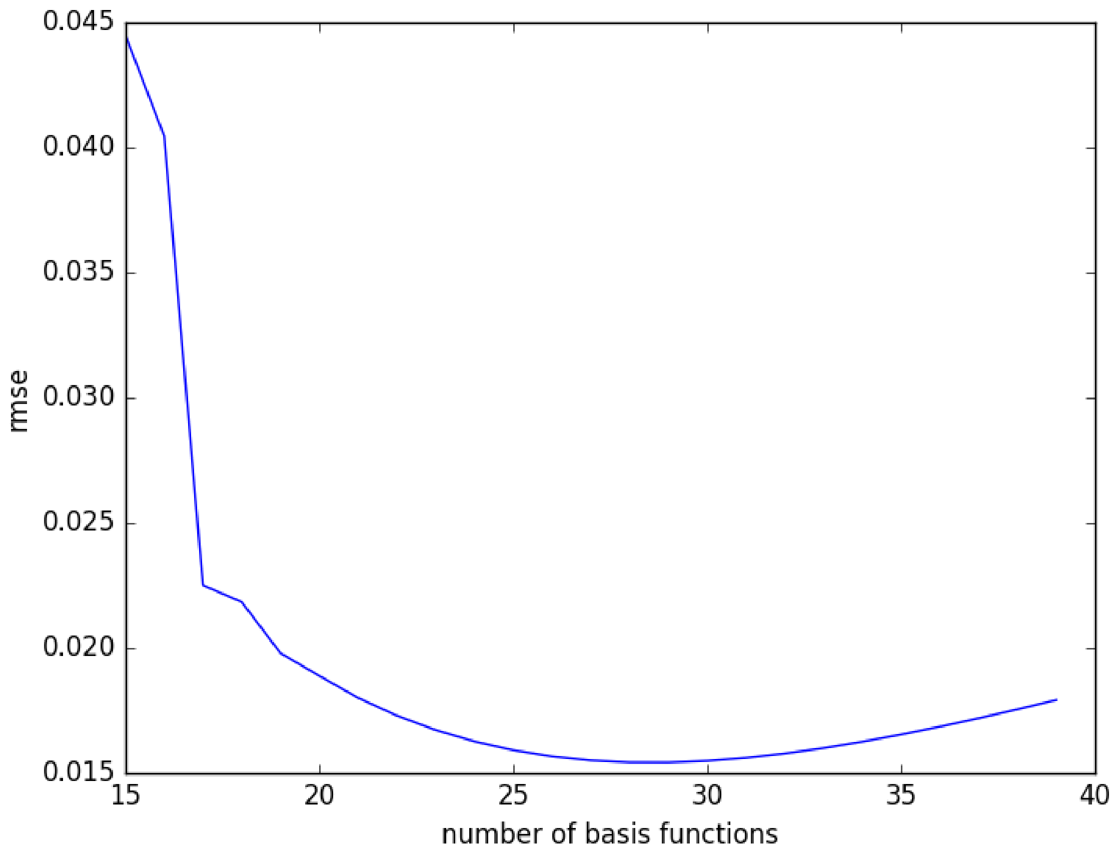
```
var = 0.02

space = np.linspace(0, 2, num=20)
x = np.linspace(0, 2, num=200)

phi = get_phi_gaussians(space, x, var)
phi = normalize(phi)
plot_phi(phi, x)
```

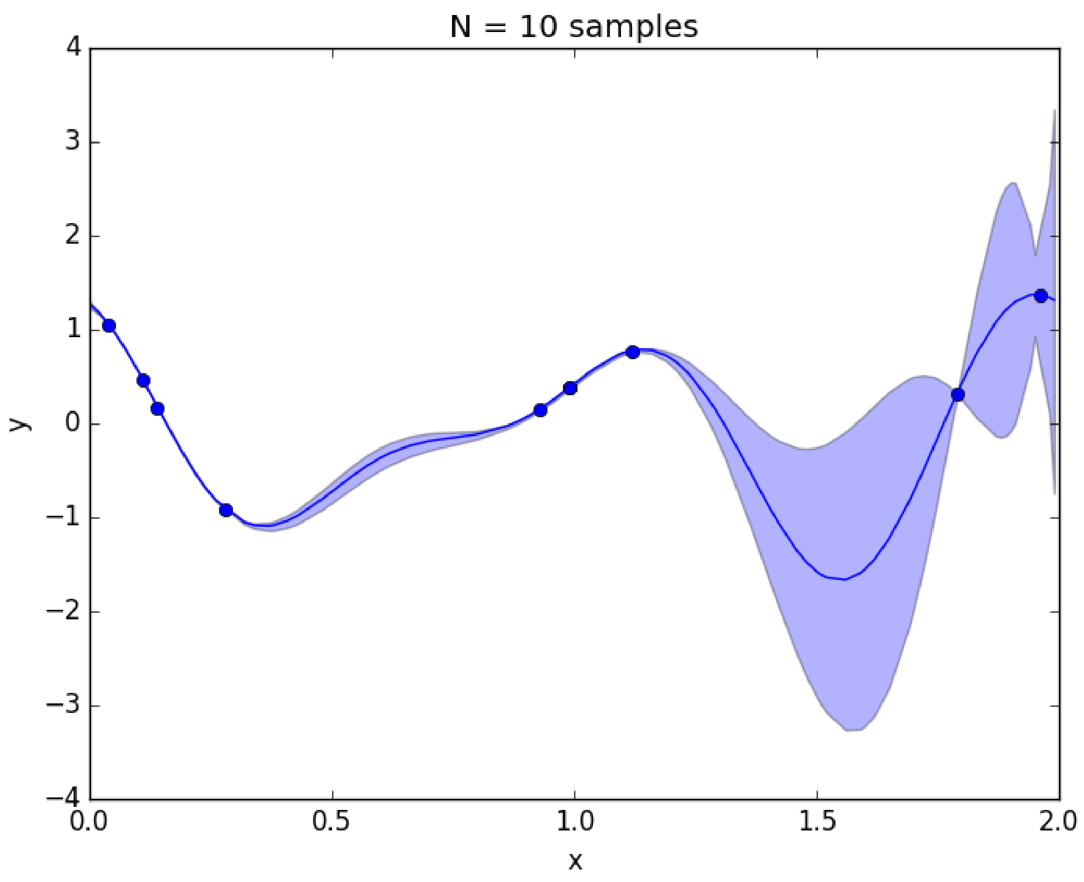
c) **Gaussian Features, Continued [6 Points]**
Repeat the process of fitting the model using the Gaussian features from the previous question. Compare the RMSE on the testing data using 15...40 basis functions and plot the RMSE. Which number of basis functions has the best performance and what is the best RMSE? Use a Ridge coefficient of $\lambda = 10^{-6}$.

The best RMSE is 0.015431 using 29 basis functions.

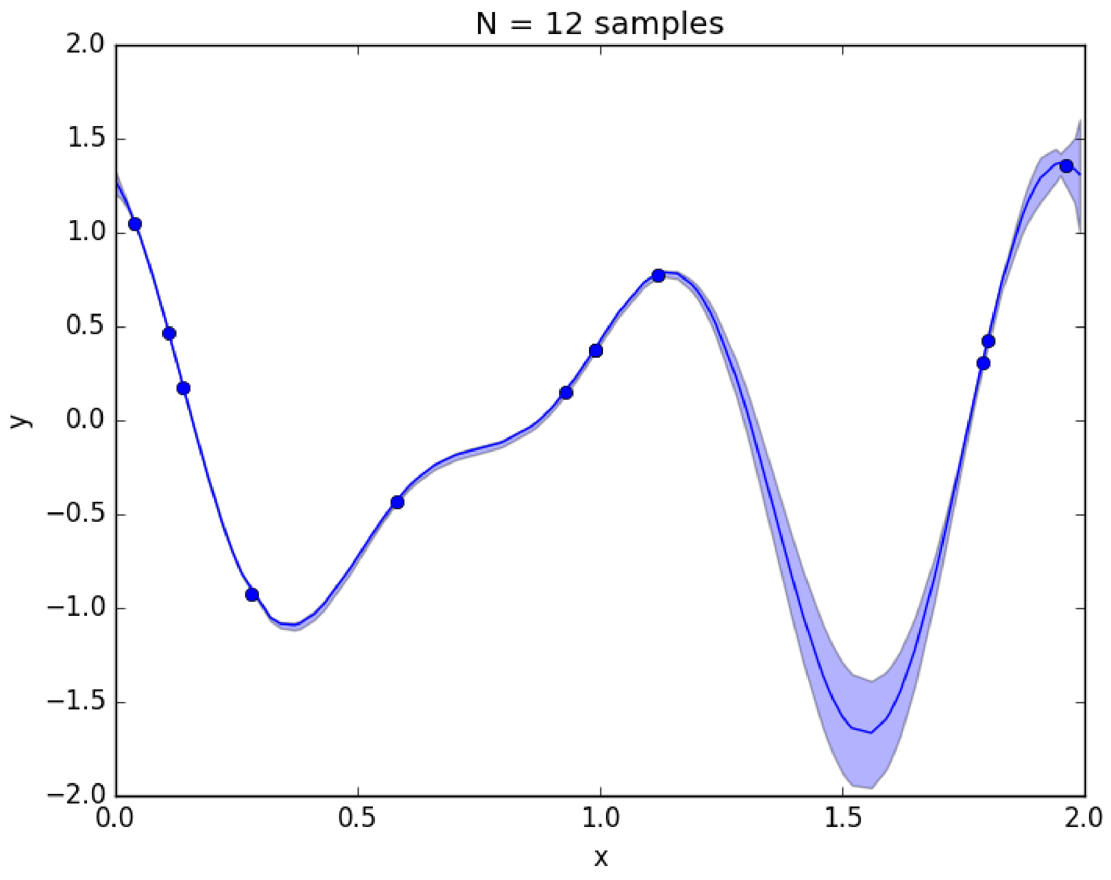


d) **Bayesian Linear Regression [10 Points]**
Using Bayesian linear regression and polynomial features of 12th degree, plot the mean and the standard deviation of the predictive distribution for each case, using the first $N = 10, 12, 16, 20, 50, 150$ data points. Discuss how the model uncertainty change with the amount of data points and the problem of overfitting with Bayesian linear regression. Use a prior $\sigma^2 = 0.0025$.

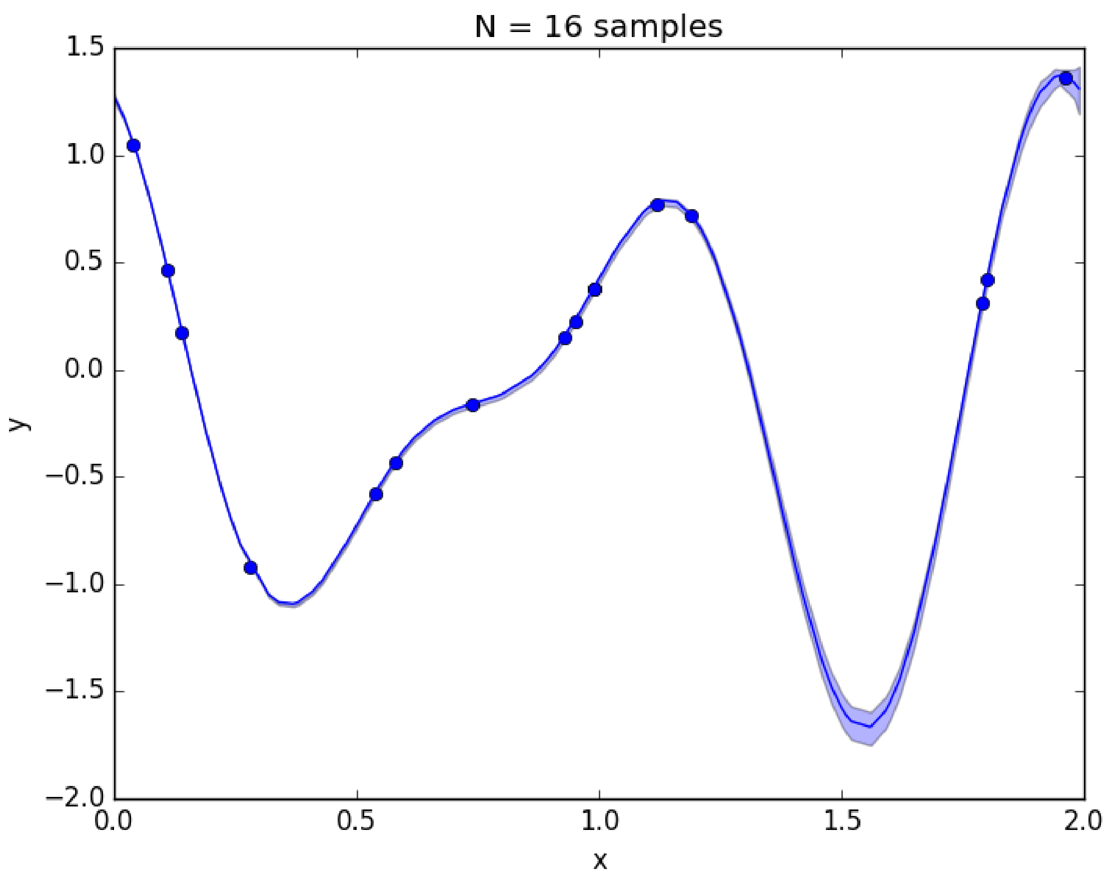
The uncertainty is getting lower the more data points we use for learning. This can be seen especially in the plot for $N = 10$, where the area between data points is very uncertain, due to the lack of available information. However, the more points we have the less uncertain we are about our prediction. For infinite data-points the uncertainty would converge to zero.
Regarding overfitting the bayesian linear regression performs better than MLE, because instead of optimizing (which leads to a single solution) we are averaging over many different solutions, which will typically lead to better predictions.



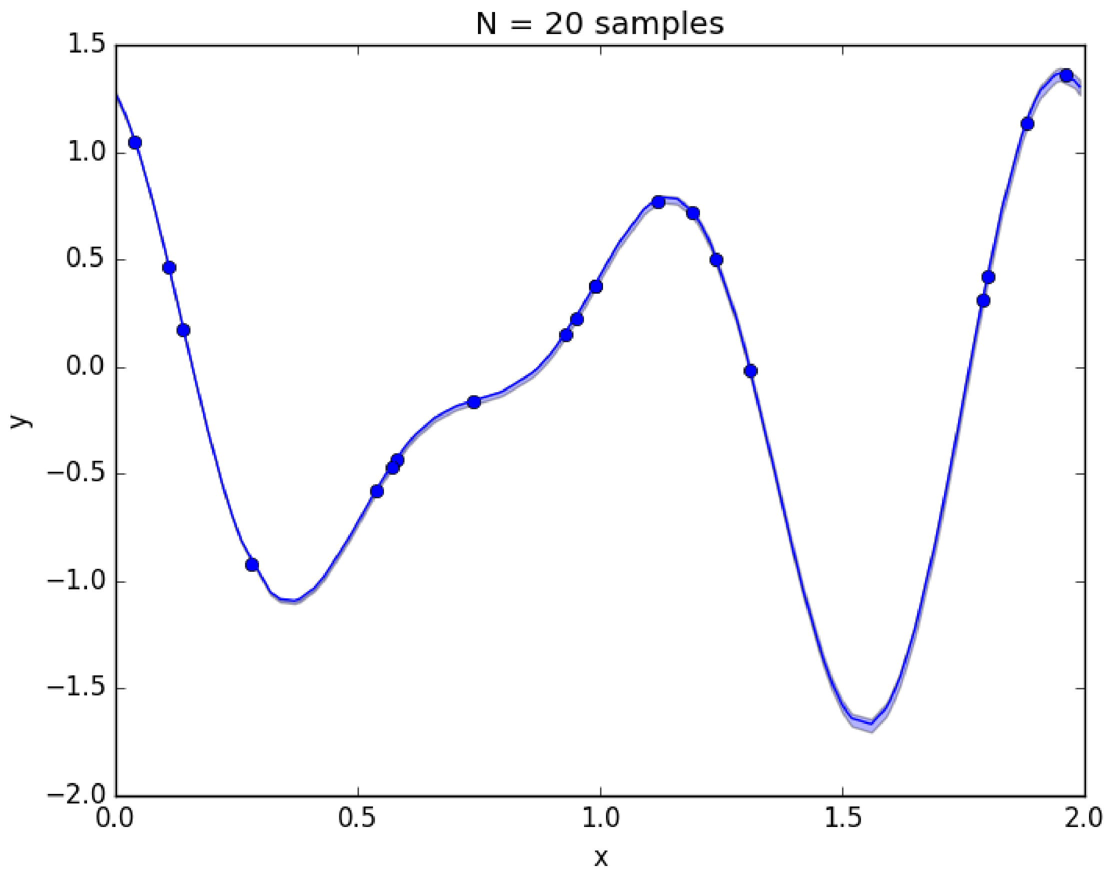
Name, Vorname: ______________________ Matrikelnummer: ␣␣␣␣␣␣␣



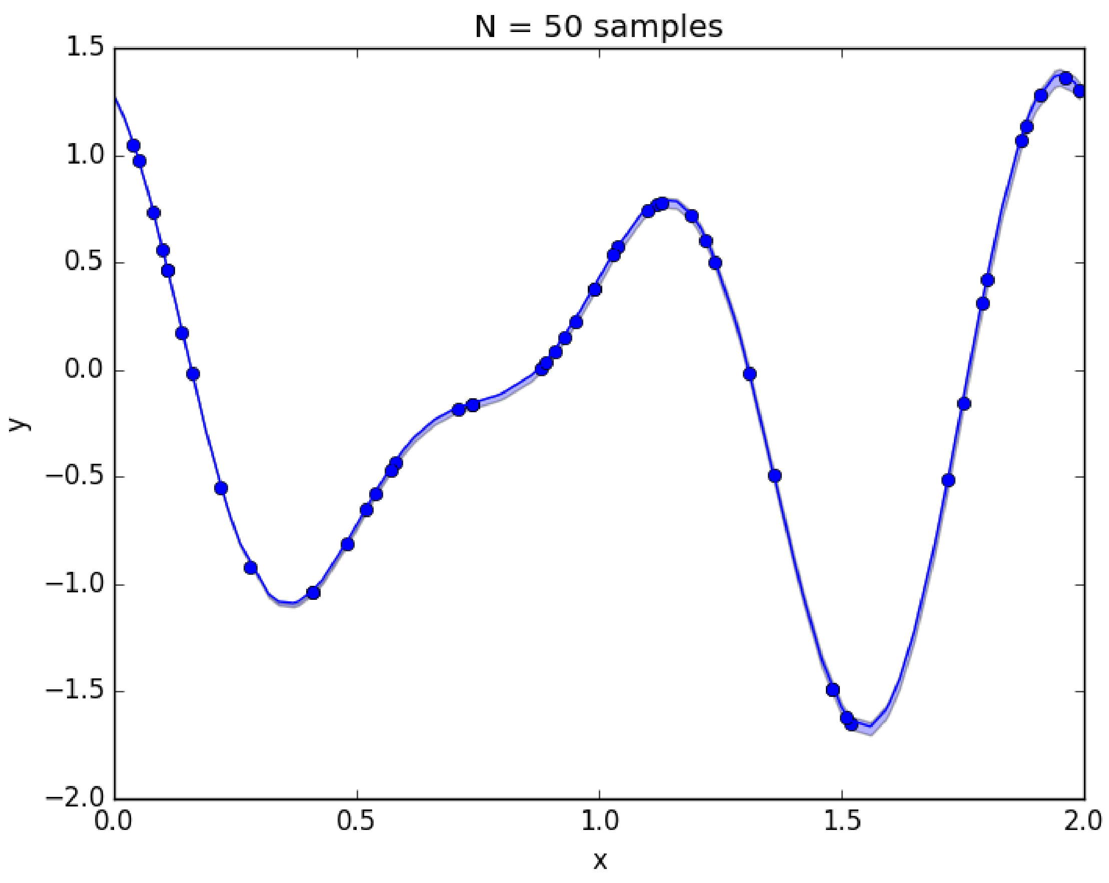
Name, Vorname: ______________________ Matrikelnummer: ⌴⌴⌴⌴⌴⌴⌴



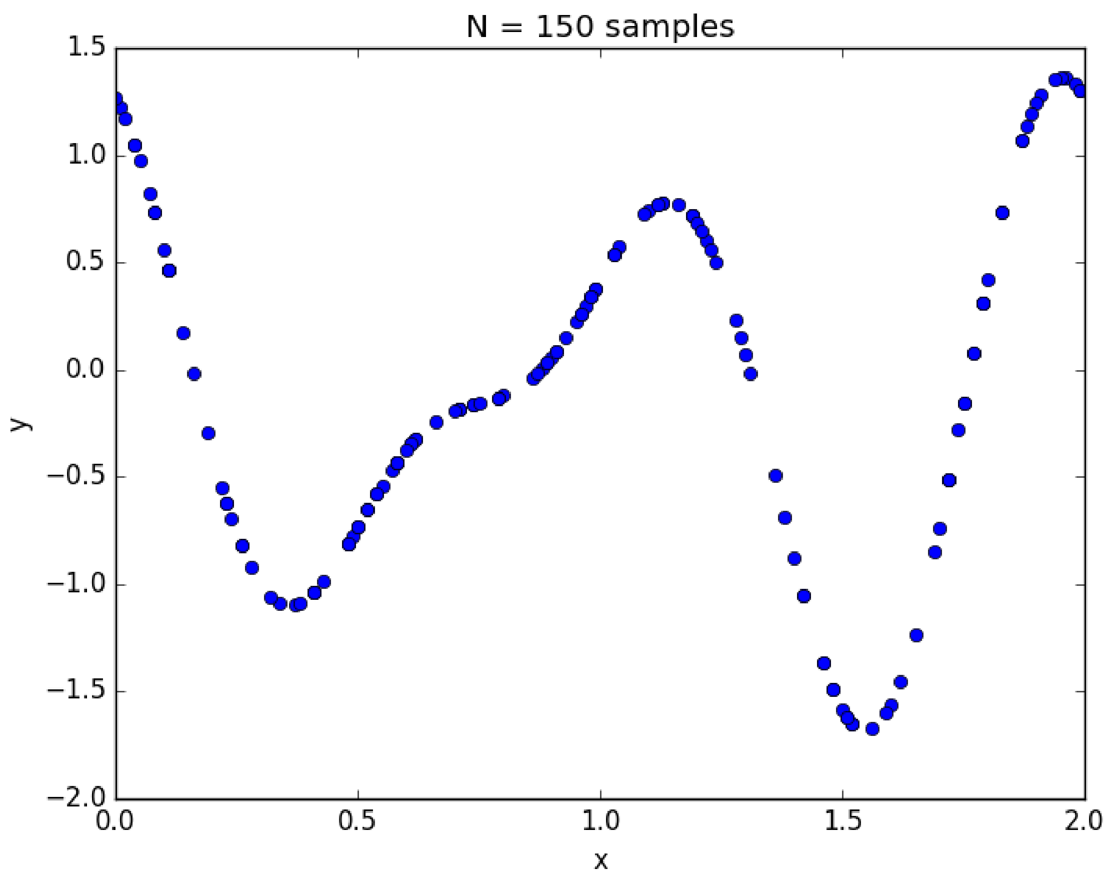
Name, Vorname: ______________________ Matrikelnummer: └┘└┘└┘└┘└┘└┘└┘



Name, Vorname: ________________________________ Matrikelnummer: ⌴⌴⌴⌴⌴⌴⌴



Name, Vorname: ______________________ Matrikelnummer: ⌴⌴⌴⌴⌴⌴⌴



e) **Bayesian Linear Regression, Continued [5 Points]**
How can we further reduce the uncertainty? Is it always a good practice?

In order to further reduce uncertainty we can use more data points. This is not always good practice, because using up to all data points defeats the purpose of a model for prediction.

f) **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.

When the original dataset is simply split into training- and test-set has the drawback, that evaluations based on this approach tend to reflect the particular characteristics of the splits. One may use statistical sampling to get better results. There are two cross-validation strategies:
***k-fold cross-validation**: The data is divided into k different chunks (of same size), which are each used once as the learning-set in k runs. Validation takes place on the respective training-sets in each run.*
*This method can be improved by choosing the k chunks in a way, that data-points are similarly distributed in all chunks. This is called **stratified k-fold cross-validation** and reduces the predictions variance.*
***Leave-one-out validation**: The data is divided in N chunks of data, one for each data-point. This results in N runs, which can lead to severe runtimes. In addition a stratified version is not possible. Furthermore there are extreme cases, in which the leave-one-out validation can give faulty results.*

Problem 3.2 Linear Classification [16 Points]

In this exercise, you will use the dataset `ldaData.txt`, containing 150 feature points $\mathbf{x}$. The first 50 points belong to class C_1, the second 50 to class C_2, the last 50 to class C_3.

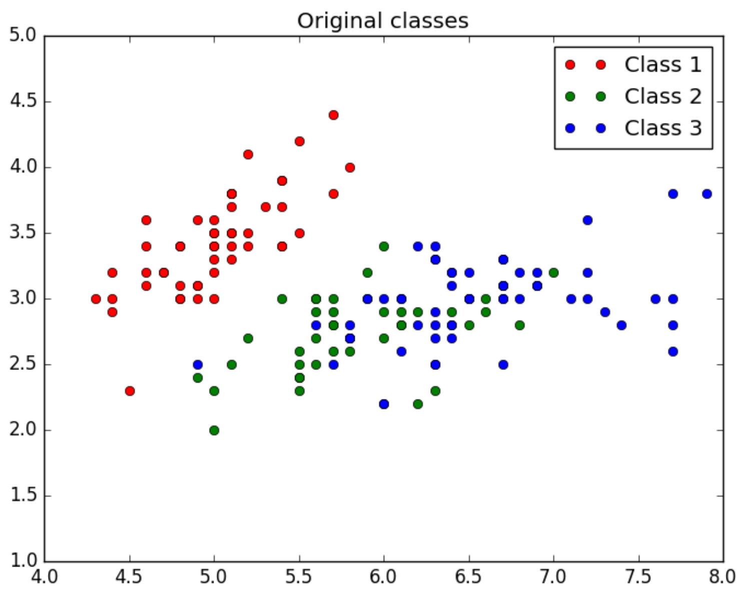
a) **Discriminative and Generative Models [4 Points]**
Explain the difference between discriminative and generative models and give an example for each case. Which model category is generally easier to learn and why?

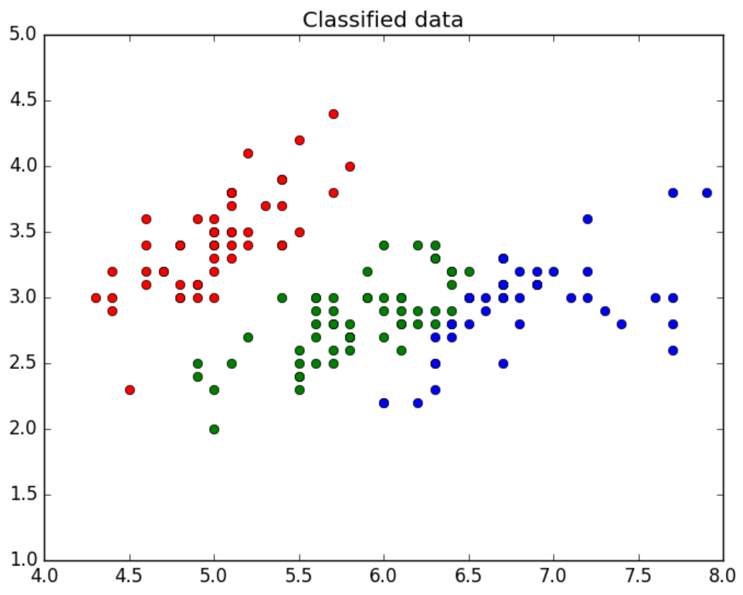
Discriminative models don't directly calculate an underlying distribution. Instead, they use discriminant functions and the resulting class depends on the result of this function on a given point x. Discriminative models are easier to learn than generative models, because they depend less on details of a given distribution.

Generative models calculate the distribution of each class. This makes them more general, but they require more calculation steps.

Name, Vorname: ______________________________ Matrikelnummer: ⊔⊔⊔⊔⊔⊔⊔

b) **Linear Discriminant Analysis [12 Points]**
Use Linear Discriminant Analysis to classify the points in the dataset. Attach two plots with the data points using 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.)





```
# Calculate direction of separating axis using fisher discriminant analysis
def fisherw(C1, C2):
    Sw = (np.cov(C1) + np.cov(C2)).T
    m_d = np.array([np.mean(C1, 1) − np.mean(C2, 1)]).T
    w = np.linalg.solve(Sw, m_d)
    w = np.divide(w, np.linalg.norm(w)).T
    return w
```

Problem 3.3 Principle Component Analysis [24 Points + 15 Bonus]

In this exercise you will use the `Iris.txt` dataset. 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. A label of 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, eigevalues and eigenvectors.)

a) **Data Normalization [4 Points]**
Normalizing the data is a common practice in machine learning. Normalize the provided dataset such as it has zero mean and unit variance per dimension. Why is normalizing important? Attach a snippet of your code.

Subtracting the mean of the dataset is important, because otherwise the results of the Principal component analysis would be meaningless. Normalizing for unit variance prevents variables with a large variance to influence the PCA too significantly.

The normalization is done by the following function:

```
def normalize(X):
    # Subtract mean
    m = np.array([np.mean(X, 0)])
    m = np.repeat(m, X.shape[0], 0)
    X = np.subtract(X, m)

    # Devide by standard deviation
    dev = np.array([np.std(X, 0)])
    X = np.divide(X, dev)
```

b) **Principle Component Analysis [8 Points]**
Apply PCA on your normalized dataset and generate a table showing the proportion (percentage) of 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.

The Principal Component Analysis of a normalized dataset X can be computed using the following snippet:

```
def pca(X):
    covar = np.cov(X, rowvar=False)
    [lambdas, evec] = np.linalg.eig(covar)
    return lambdas, evec
```

The cumulative explained variance of these Eigenvalues is

λ_1	λ_2	λ_3	λ_4
0.72770452	0.95800975	0.99484807	1.

Therefore, in order to explain more than 95% variance, 2 Eigenvectors are needed.

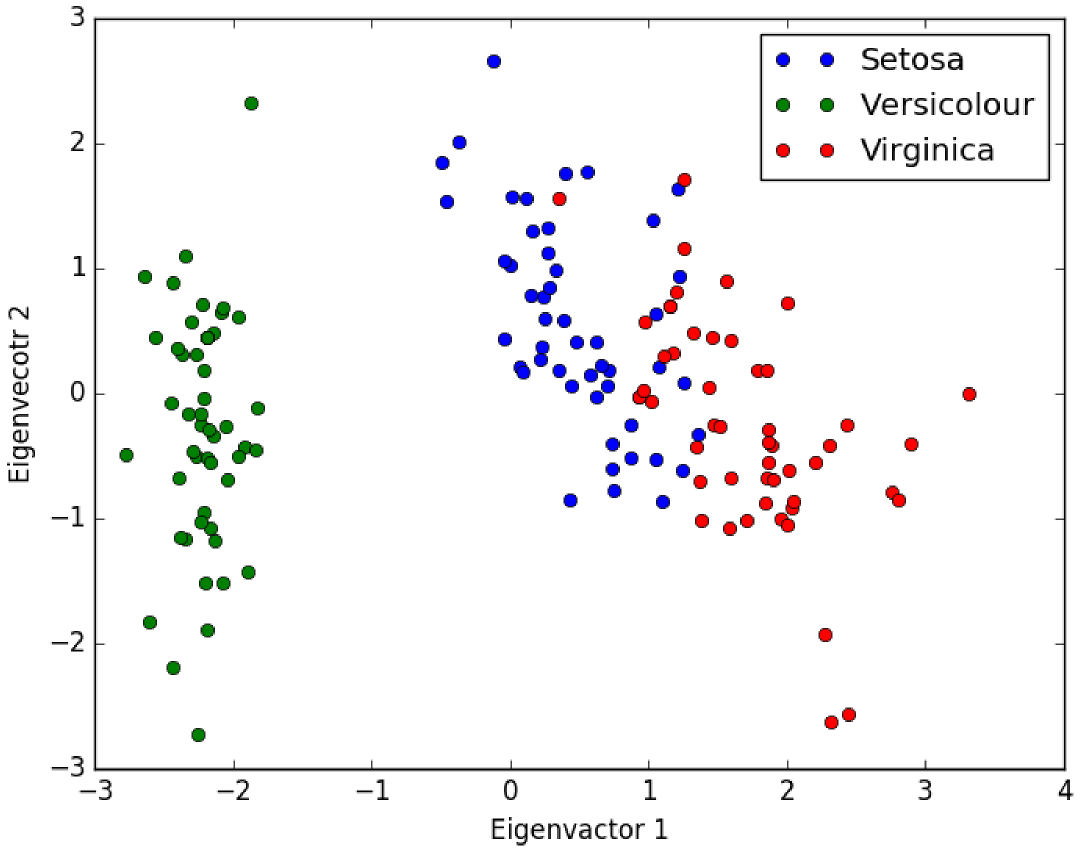
c) **Low Dimensional Space [6 Points]**
Using as many components as needed to explain 95% of the dataset variance, generate a scatter plot of the lower-dimensional projection of the data. Use different colors or symbols for data points from different classes. What do you observe? Attach a snippet of your code.

The first step is to project the data. This is done by a simple dot-multiplication. In the snippet, the dataset is transposed because of the order of the data in the given file. The result is transposed to restore that order:

```
# Project data onto principal components
datap = np.dot(evec.T, data.T).T
```

The data is plotted using the following snippet, where C0, C1 and C2 indicate the samples that belong to the corresponding class:

```
# Plot result
plt.plot(datap[c0, 0], datap[c0, 1], 'o', label='Setosa')
plt.plot(datap[c1, 0], datap[c1, 1], 'o', label='Versicolour')
plt.plot(datap[c2, 0], datap[c2, 1], 'o', label='Virginica')
plt.xlabel("Eigenvactor␣1")
plt.ylabel('Eigenvecotr␣2')
plt.legend(loc='best')
plt.show()
```



From this perspective, the plant Versicolour is easily separable from the other two plants. Setosa and Virginica are also separated, but they overlap.

d) **Projection to the Original Space [6 Points]**
Reconstruct the original dataset by using different number of principle components. Using the normalized root mean square error (NRMSE) as a metric, create a table with the error per input versus the amount of principle components used (i.e., with five columns: first for the number of components, remainder for the input NRMSE). Attach a snippet of your code. (Remember that in the first step you normalized the data.)

Since the data has been normalized in the beginning, the NRMSE is calculated automatically by projecting the data into the original space and then calculating the RMSE. This is done by the following script:

```
for i in np.arange(1, 5):
    # Set all unused principal components to 0
    data_restored = np.copy(datap)
    data_restored[:, i:] = 0

    # Project features back to original feature space
    data_restored = np.linalg.solve(evec.T, data_restored.T).T

    # Calculate RMSE
```

```
e = np.subtract(data, data_restored)
e **= 2
e = np.sum(e, axis=1)
e = np.divide(np.sum(e, axis=0), e.size)
e = np.sqrt(e)
print e
```

The result is shown in the following table. As expected, the value drops significantly, for a high number of components.

Components	1	2	3	4
NRMSE	*1.043*	*0.4098*	*0.1435*	$3.002 \cdot 10^{-16}$

e) **Kernel PCA [15 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 linearly 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 much as concise (but clear) as possible.

The PCA decorrelates the features given in a dataset by diagonalizing the covariance matrix, given by

$$S = \frac{1}{N}\sum_{n=1}^{N} x_n x_n^T$$

Unfortunately, this only works well, if the relation between the feature is linear.

To circumvent this problem, a transformation $\Phi(x)$ can be used to project the samples into a feature space. In an ideal feature space, the relations between two different features are only linear. Consider an example, where two features, $x_{(1)}$ and $x_{(2)}$ have a quadratic relationship: $x_{(2)} = 5 \cdot x_{(1)}^2$. A transformation $\Phi(x) = \begin{pmatrix} x_{(1)}^2 \\ x_{(2)} \end{pmatrix}$ would make this relationship linear and therefore PCA applicable.

This means, that in Kernel PCA, the covariance matrix of the feature space

$$S' = \frac{1}{N}\sum_{n=1}^{N} \Phi(x_n)\Phi(x_n)^T$$

is diagonalized by solving its eigenvector equation

$$S' v_i = \lambda_i v_i$$

However, transforming every sample into the feature space by applying $\Phi(x)$ is can be a very computation intensive operation. Therefore, the goal is to solve this eigenvector problem without having to work in the feature space. This is avoided, using the kernel trick. Instead of calculating $\Phi(x_n)$ and $\Phi(x_m)$ explicitly, it is enough to calculate $k(x_n, x_m) = \Phi(x_n)\Phi(x_m)$. Typically, this is significantly faster, since $\Phi(x)$ can have a very high dimensionality.