3rd Session

Martin Biehler

Aug 03, 2021

CONTENTS

```
Der Input diese Skripts ist in der Vorbereitung eines Semiars entstanden, welches ich␣
↪im Sommersemester 2021 zum ersten Mal an der FHNW gegeben habe.
```

good python code examples: https://www.inovex.de/de/blog/support-vector-machines-guide/

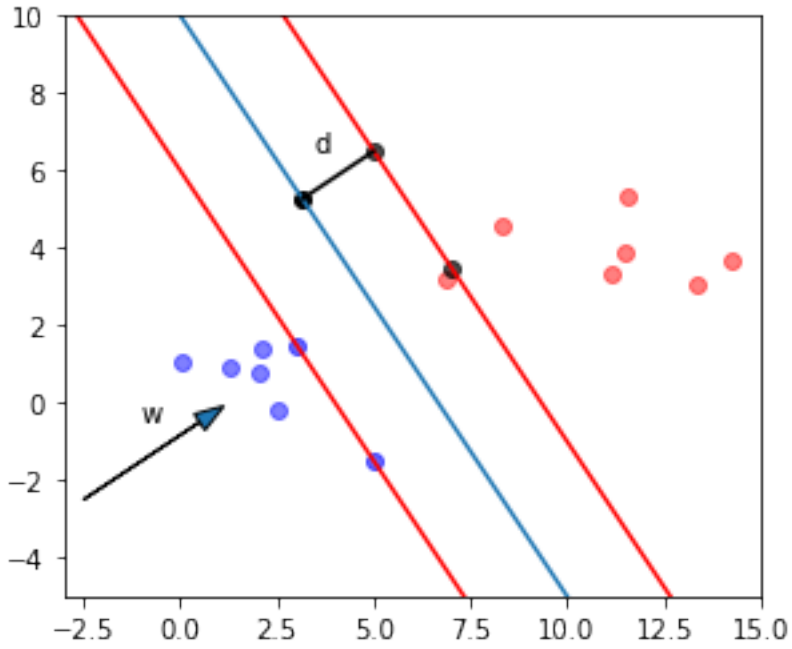
CHAPTER

ONE

SUPPORT-VECTOR-MACHINES = MAXIMUM MARGIN CLASSIFICATION

```
import numpy as np
from IPython.display import Image
import matplotlib.pyplot as plt
```

```
Text(-1, -0.5, 'w')
```



Im 2-dimensionalen Raum haben wir die Variablen x, y. Alle Punkte auf einer Geraden (separating plane) bekommt man:

$$a + b \cdot x = c \cdot y$$

Trivialerweise ist $c = 1$ meist gültig. Dies entspricht exakt unserer Geraden-Gleichung aus der Linearen Regression. Umgekehrt kann man die Gerade auch als die Menge aller Punkte definieren, die folgender Bedingung genügen:

$$a + b \cdot x + c \cdot y = 0$$

Im trivialen Fall wäre $c = -1$

Für alle Punkte unterhalb der Gerade gilt:

$$a + b \cdot x + c \cdot y < 0$$

und für alle Punkte oberhalb der Geraden gilt:

$$a + b \cdot x + c \cdot y > 0$$

Für die Klassification gilt, dass die Klassen immer mit -1 und 1 benannt werden. Ein neuer Punkt (x, y) wird klassifiziert:

$$\hat{\text{pred}} = \text{sign}(a + b \cdot x + c \cdot y)$$

1.1 Aenderung der Schreibweise:

In der Literatur zu Support Vector Machines werden die Koeffizienten anders benannt als in der klassischen Linearen Regression. Anstatt a für den intercept wird b verwendet. Anstatt b für den Steigungskoeffizienten wird ω verwendet:

$$\hat{\text{pred}} = \text{sign}(b + \omega_1 \cdot x_i + \omega_2 \cdot x_2)$$

und in Vektorschreibweise (exakt identisch zur Multivariaten Linearen Regression - mit Ausnahme der Bennenung der Koeffizienten):

$$\hat{\text{pred}} = \text{sign}(b + \boldsymbol{\omega x})$$

Es kann gezeigt werden, dass der Vektor $\boldsymbol{\omega}$ senkrecht auf die Trennungs-Ebene (in der Graphik oben die mittlere Gerade) steht. Dies habe ich als Pfeil eingezeichnet.

Nun soll eine Ebene gefunden werden, deren **margin maximal** ist; als margin wird der Abstand zu den nähesten Daten-Punkten bezeichnet. Die Datenpunkte, die auf den margins zu liegen kommen heissen **support vectors**. Die Enfernung dieser Punkte von der Trennlinie kann angegeben werden:

$$d = y_i(\boldsymbol{\omega x}_i + b); \qquad y_i \in \{-1, 1\},$$

hier ändert y_i das Vorzeichen so, dass die Distanz immer positiv ist.

Für alle Punkte $\mathbf{x_i}$, die genau die Distanz 1 haben gilt:

$$\begin{aligned} 1 &= y_i(\boldsymbol{\omega x}_i + b) \\ \frac{1}{\|\boldsymbol{\omega}\|} &= \left|\frac{\boldsymbol{\omega x}_i}{\|\boldsymbol{\omega}\|} + \frac{b}{\|\boldsymbol{\omega}\|}\right| \end{aligned}$$

$\|\boldsymbol{\omega}\|$ ist die Länge des Vektors $\boldsymbol{\omega}$, die sogenannte Vecktor-Norm. Teile ich einen Vektor durch seine Vektor-Norm, dann hat er genau die Länge 1. Weil alle Punkte auf dem margin dieselbe Distanz zur Trennlinie haben und weil der margin symmetrisch ist muss

$$p = \frac{2}{\|\boldsymbol{\omega}\|}$$

maximiert werden, um den margin möglichst gross zu machen. Das bedeutet, dass $\|\boldsymbol{\omega}\|$ möglichst klein sein muss. Der Abstand der beide support-vectors ist also indirekt proportional zur Länge des Vectors ω. Als Minimierungs-Problem ergibt sich:

$$(\omega^*, b^*) = \text{argmin}\frac{1}{2}\|\boldsymbol{\omega}\|$$

Maximieren von $\frac{2}{\|\boldsymbol{\omega}\|}$ ist gleichbedeutend mit Minimieren von $\frac{\|\boldsymbol{\omega}\|}{2}$. Damit haben wir die Formulierung unseres maximum-margin classifiers: Finde $\boldsymbol{\omega}$ und b so, dass:

1. $\frac{1}{2}\boldsymbol{\omega}^T\boldsymbol{\omega}$ minimiert wird, für alle
2. $(\mathbf{x}_i, y_i), y_i(\boldsymbol{\omega x}_i + b) \geq 1$

Weil die Vektor-Norm sich als die Quadratwurzel der aufsummierten quadrierten Vectorelemente berechnet:

$$\|\boldsymbol{\omega}\| = \sqrt{\omega_1^2 + \omega_2^2 + ... \omega_d^2}$$

kann man auch schreiben:

$$\|\boldsymbol{\omega}\| = \sqrt{\boldsymbol{\omega}^T \boldsymbol{\omega}}$$

Da die Quadratwurzel eine monotone Transformation ist, kann sie in unserem Minimierungs-Problem einfach weggelassen werden. Somit ergibt sich obige 1. Bedinung.

Will man das lösen, ergibt sich ein quadratische Optimierungsproblem, welches mit Lagrange Multiplikatoren gelöst werden kann.

Hierbei ergibt sich die characteristische Gleichung (dual form) zu:

$$\max_\alpha \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \cdot \mathbf{x}_j$$

Für das Verständnis der nachfolgenden Erklärungen zu kernels, ist der Ausdruck $\mathbf{x}_i^T \cdot \mathbf{x}_j$ entscheidend. Hierbei wird das Skalarprodukt (dot product) eines jeden Daten-Punktes $\boldsymbol{x}_i$ mit jedem anderen Daten-Punkt $\boldsymbol{x}_j$ berechnet. Für normierte Vektoren $\mathbf{x}_i$ ist das Skalarprodukt ein Ähnlichkeitsmass (cosine-similarity, correlation-coefficient).

1.2 Problem:

Der Ausdruck

$$\mathbf{x}_i^T \cdot \mathbf{x}_j$$

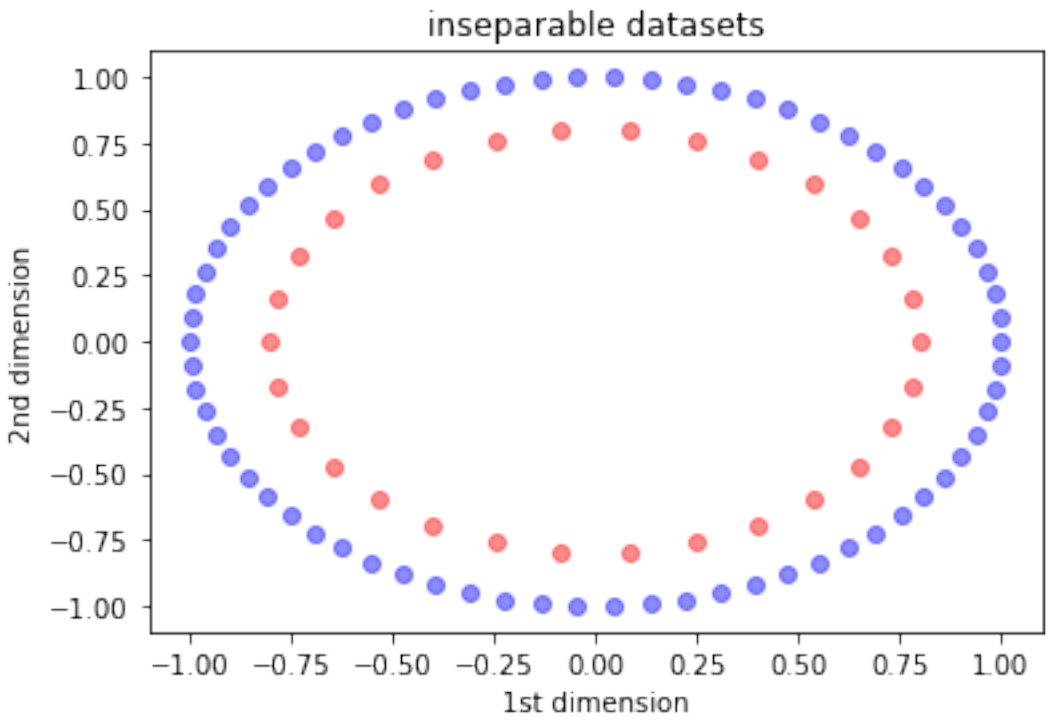
in obiger Gleichung zeigt, dass Support-Vecktor-Machines quadratisch mit der Anzahl der Trainings-Punkte skalieren. SVMs sind also für BigData nicht zu gebrauchen.

1.3 Kernel-Trick

```
from sklearn.datasets import make_circles
import matplotlib.pyplot as plt
import numpy as np
```

```
x, y = make_circles(n_samples = (70, 30), random_state=42, factor=0.8)
```

```
Text(0, 0.5, '2nd dimension')
```



In the above Figure, the two data-sets (red dots and blue dots) are not linearily separable. There is no plane/line that can separate the two classes.

1.3.1 project the data into 3 dimensions

$$\Phi(\mathbf{x}) \rightarrow \mathbf{x}_1^2, \mathbf{x}_2^2, \sqrt{2}\mathbf{x}_1\mathbf{x}_2,$$

i.e. there is a transformation that projects the x- and y-coordinates to new x- and y-coordinates and to an additional z-coordinate.

```
# this is the projection into the 3rd coordinate (z-coordinate)
x3 = np.sqrt(2) * x[:, 0] * x[:, 1]
x3 = x3.reshape((-1, 1))
# in the first column of x we have the x-coordinates, in the second column the y-
 ↪coordinate
mapped_x = np.concatenate((x ** 2, x3), axis=1)
# mapped_x has three coordinates right now:
mapped_x.shape
```

```
(100, 3)
```

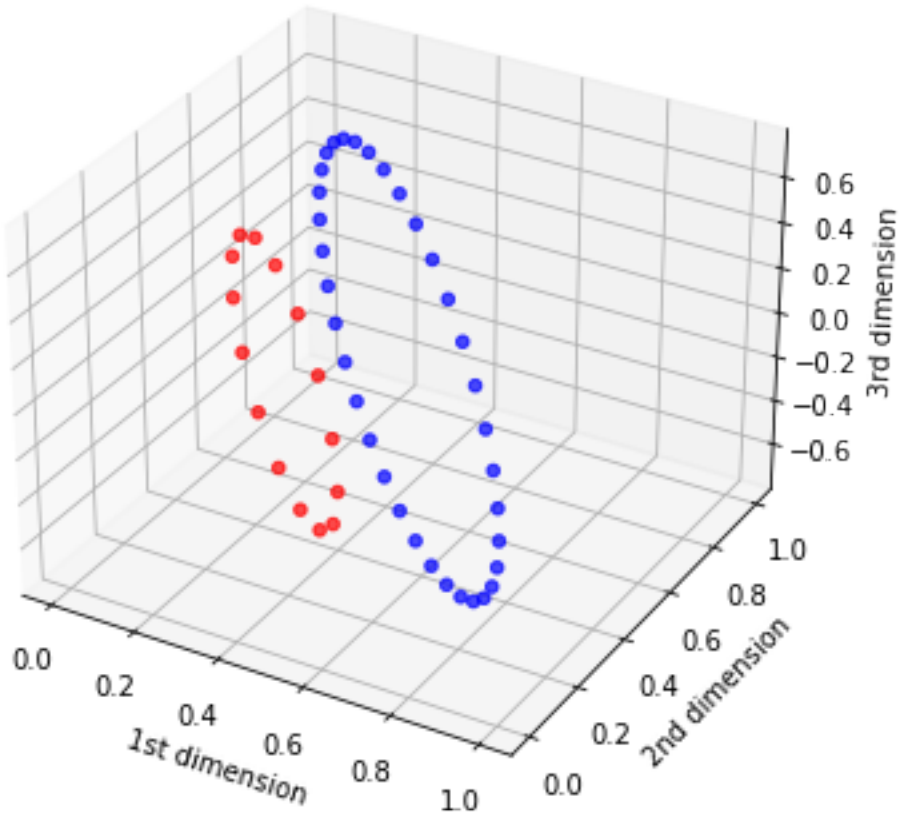
```
fig = plt.figure(figsize = (8,6))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(mapped_x[y==1, 0], mapped_x[y==1, 1], mapped_x[y==1, 2], color='red',␣
 ↪alpha = 0.45)
ax.scatter(mapped_x[y==0, 0], mapped_x[y==0, 1], mapped_x[y==0, 2], color='blue',␣
 ↪alpha = 0.45)
#plt.title('inseparable datasets')
ax.set_xlabel('1st dimension')
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel('2nd dimension')
ax.set_zlabel('3rd dimension')
```

```
Text(0.5, 0, '3rd dimension')
```



As can be seen, the additional 3rd dimension makes the problem far easier to solve. Now, a plane can easily separate the two classes.

1.3.2 just another projection into 3D

$$\Phi(\mathbf{x}) \rightarrow \mathbf{x}_i, \mathbf{x}_2, \sum_i^n \exp\left(-\frac{\|\mathbf{x}-\mathbf{x}_i\|^2}{2\sigma^2}\right)$$

BUT as we will see later, there exists no kernel-trick for this projection. The projection is helpfull to make the two classes separable but other projections allow for more efficient solutions.

```
# not 100 % correct though
# https://stats.stackexchange.com/questions/63881/use-gaussian-rbf-kernel-for-mapping-
↪of-2d-data-to-3d

from scipy.spatial.distance import pdist, squareform
from scipy import exp
```

(continues on next page)

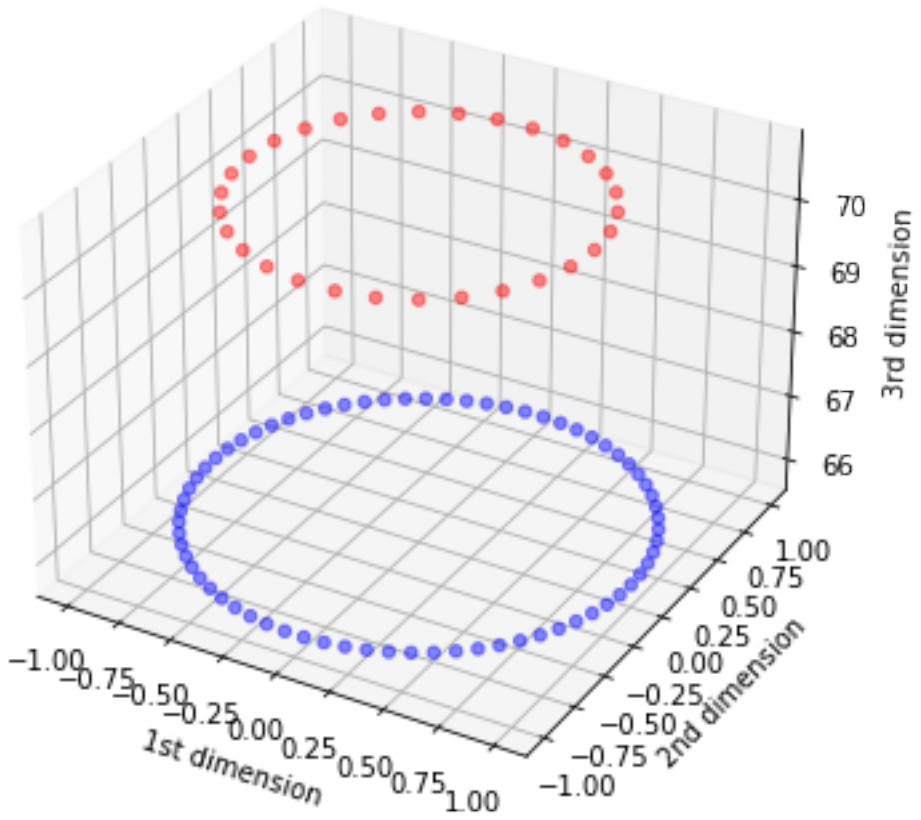
(continued from previous page)

```
pairwise_d = squareform(pdist(x, 'euclidean'))
projection = exp(-pairwise_d **2 / (2 ** 2))
new_dim = np.sum(projection, axis=1)
```

```
/home/martin/miniconda3/envs/book/lib/python3.7/site-packages/ipykernel_launcher.
 ↪py:7: DeprecationWarning: scipy.exp is deprecated and will be removed in SciPy 2.0.
 ↪0, use numpy.exp instead
  import sys
```

```
fig = plt.figure(figsize = (8,6))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x[y==1, 0], x[y==1, 1], new_dim[y==1], color='red', alpha = 0.45)
ax.scatter(x[y==0, 0], x[y==0, 1], new_dim[y==0], color='blue', alpha = 0.45)
#plt.title('inseparable datasets')
ax.set_xlabel('1st dimension')
ax.set_ylabel('2nd dimension')
ax.set_zlabel('3rd dimension')
```

```
Text(0.5, 0, '3rd dimension')
```



1.3.3 Der kernel-Trick besteht genau darin, dass die Projektion in einen höher-dimensionalen Raum nicht vorgenommen werden muss

Wir erinnern die charakteristische Formel des maximal-margin classifiers:

$$\max_\alpha \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \cdot \mathbf{x}_j$$

Anstatt mit den urspr. Daten-Punkten $\mathbf{x}_i$ und $\mathbf{x}_j$ muss diese Gleichung jetzt für die Projektionen der Daten vorgenommen werden:

1.3.4 unsere erste Projektion:

$\Phi(\mathbf{x}) \rightarrow \mathbf{x}_1^2, \mathbf{x}_2^2, \sqrt{2}\mathbf{x}_1\mathbf{x}_2$

einsetzen in das Skalarprodukt (dot-product) für 2 Daten-Punkte $\mathbf{x}_i$ und $\mathbf{x}_j$ (wie in der characteristischen Gleichung oben):

$$\begin{aligned} \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle = \\ \langle [x_{i1}^2, x_{i2}^2, \sqrt{2}x_{i1}x_{i2}], [x_{j1}^2, x_{j2}^2, \sqrt{2}x_{j1}x_{j2}] \rangle = \\ x_{i1}^2 x_{j1}^2 + x_{i2}^2 x_{j2}^2 + 2x_{i1}x_{i2}x_{j1}x_{j2} \end{aligned}$$

1.3.5 Trick: Es gibt eine einfachere Rechenvorschrift, die zum selben Ergebnis kommt. Das ist der Kernel-Trick

$$\begin{aligned} \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle = \langle \mathbf{x}_i, \mathbf{x}_j \rangle^2 = \\ \langle [x_{i1}, x_{i2}], [x_{j1}, x_{j2}] \rangle^2 = \\ (x_{i1}x_{j1} + x_{i2}x_{j2})^2 = \\ x_{i1}^2 x_{j1}^2 + x_{i2}^2 x_{j2}^2 + 2x_{i1}x_{i2}x_{j1}x_{j2} \end{aligned}$$

Man kann jetzt also schreiben:

$$k(\mathbf{x_i}, \mathbf{x_j}) = \langle \mathbf{x}_i, \mathbf{x}_j \rangle^2$$

Für unsere charackteristische Gleichung könnten wir nun schreiben:

$$\max_\alpha \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j k(\mathbf{x_i}, \mathbf{x_j})$$

Unser maximal-margin Classifier könnte also das Problem der 2 Ringe lösen, ohne explizit die Projektione in 3D vornehmen zu müssen.

Unterschied: 1.Methode:

- Projektion in höher-dimensionalen Raum
- Multiplikation in höher-dimensionalen Raum

2.Method:

- Multiplikation im Ausgangs-Raum
- Quadratur des Ergebnisses

1.4 What are kernels exactly?

Basically, kernels are similarity measures outside the usual Euclidean space. We can interpret kernels in two different ways:

Either

- we project the data into higher dimensional spaces and compute Euclidean distances (similarities)

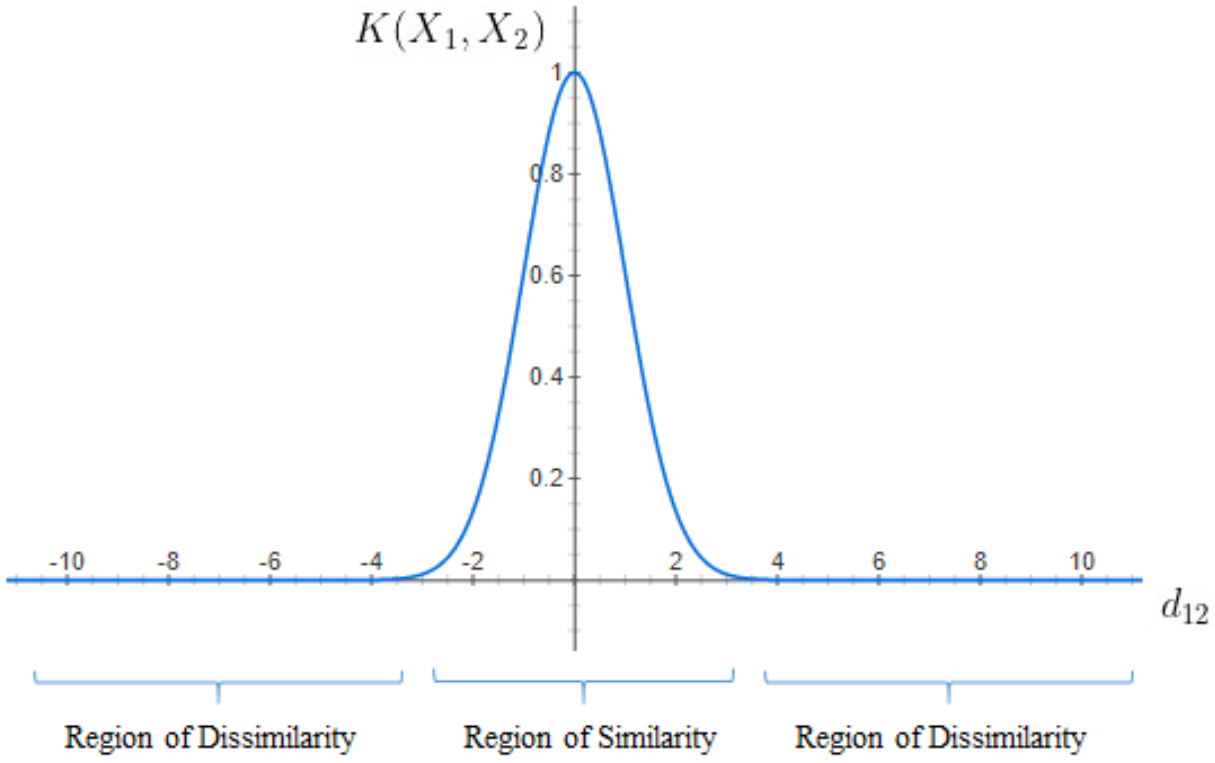
Or

- we use a different distance measure (other then Euclidean distance) in the original space that correponds to Euclidean distance in the projected dimension

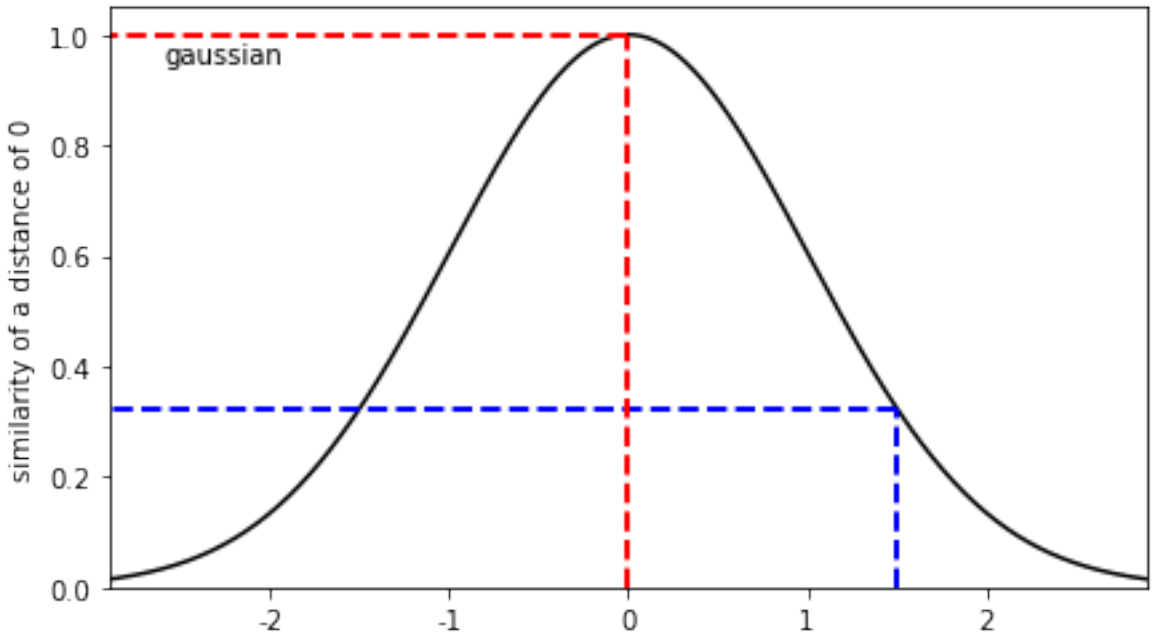
For example, take the **radial basis function kernel**:

$$K(X_1, X_2) = \exp\left(-\frac{\|X_1 - X_2\|^2}{2\sigma^2}\right)$$

$\|X_1 - X_2\|$ is the Euclidean distance between the points X_1 and X_2 or between two vectors $\mathbf{x}_1$ and $\mathbf{x}_2$. If this distance is 0, i.e. the two cases are actually identical, then the similarity - as measured by the rbf-kernel - is 1:



```
[<matplotlib.lines.Line2D at 0x7f85f4f6d190>]
```



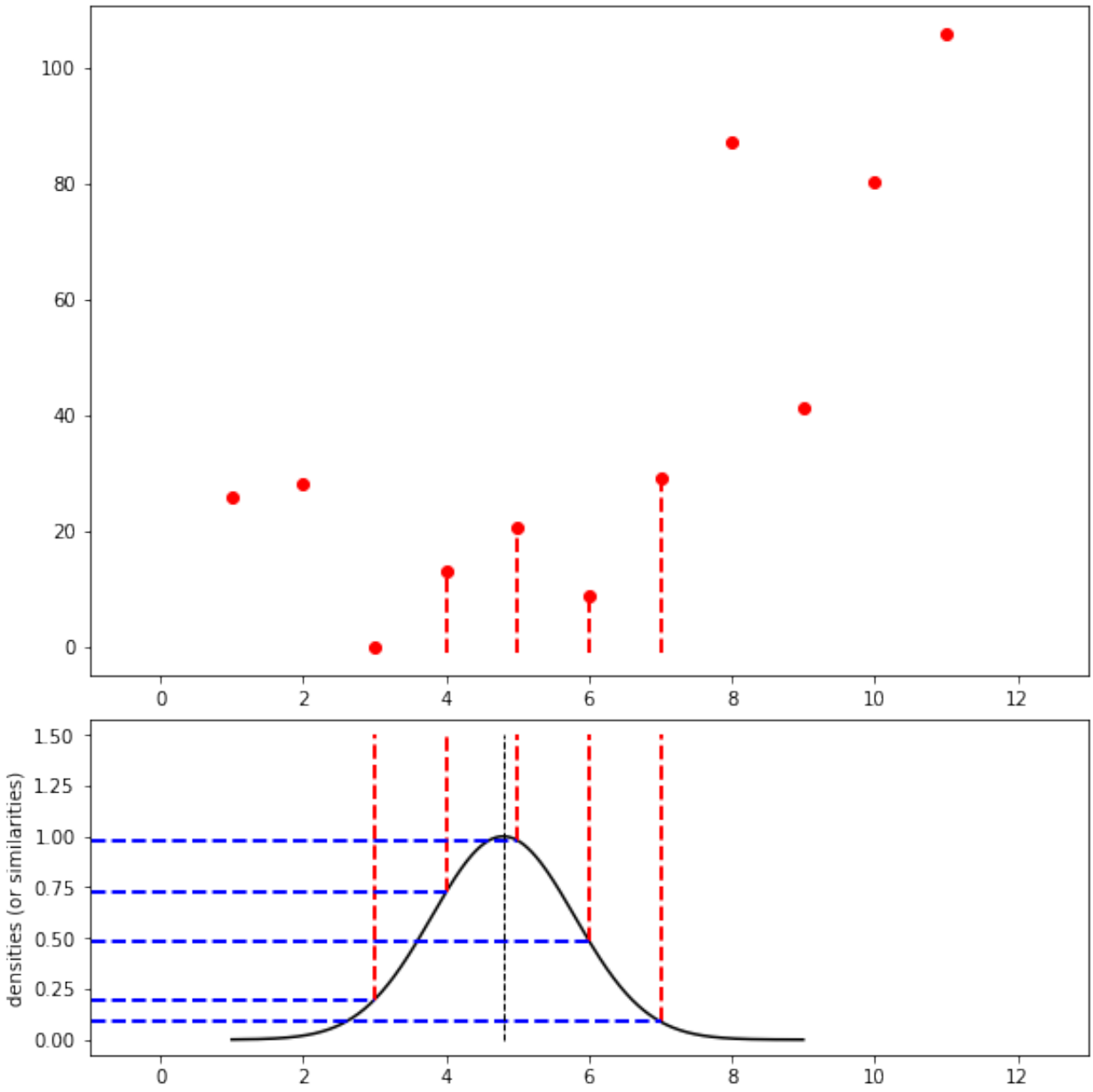
CHAPTER

TWO

SIMPLE KERNEL-REGRESSION

Here, Kernels just define a neighborhood by similarity in kernel-space. For a certain point the values of its neighbors are weighted by similarity and then averaged. The Euclidean distance on the x-axis is transformed into a similarity-measure on the y-axis.

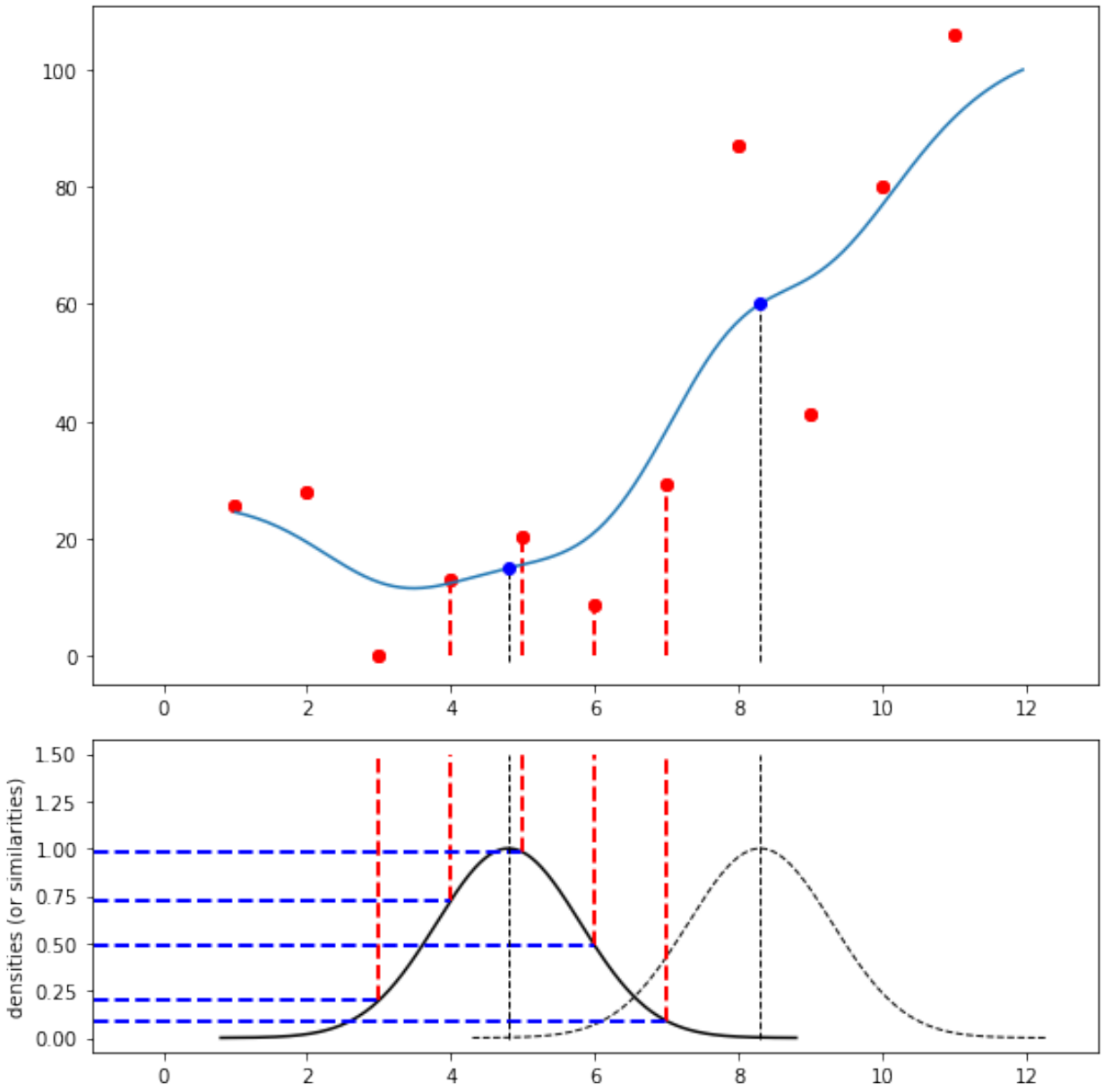
```
Text(0, 0.5, 'densities (or similarities)')
```



The prediction takes part at the center x of the kernel. The similiarities to this center-point x are evaluated for all points x_i in its vicinity and the values y_i are weighted by the respective similarity. Points closer to the center x receive higher weights and have more influence on the predicted value. In order to make the prediction independent of the exact weights we have to divide by the sum of the weights.

$$\hat{m}(x) = \frac{\sum_i^n K_h(x - x_i) y_i}{\sum_i^n K_h(x - x_i)} \tag{2.1}$$

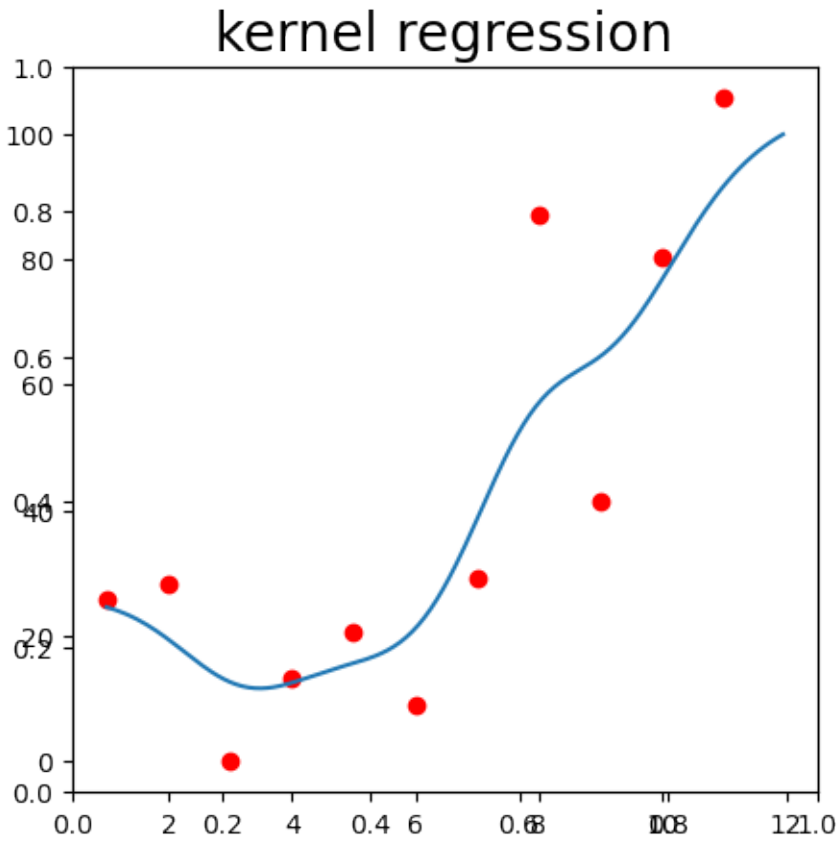
Here, h is the bandwidth or the scale of the kernel. In the context of a gaussian kernel one could also say the **variance**.




```
x = np.arange(1, 12, 0.05).reshape((-1, 1))
y_kr = []
for x_i in x:
    densities = np.exp(KernelDensity(kernel='gaussian').fit(np.array([[x_i[0]]])).
 ↪score_samples(X))
    y_kr.append(np.sum(densities.reshape((-1, 1)) * y)/sum(densities))

f = plt.figure(figsize=(5, 5), dpi=100)
plt.title(label='kernel regression', fontdict={'fontsize':20})
axes = f.add_subplot(111)
axes.plot(X[:,0], y, 'ro', x[:,0], np.array(y_kr).reshape((-1,)))
#axes = plt.gca()
axes.set_ylim([np.min(y)-5, np.max(y) +5])
```

```
(-5.0, 110.77315979942053)
```



2.1 How is the similarity-measure related to the support-vector-machine?

Remember that in the characteristic equation we had:

$$\mathbf{x}_i^T \cdot \mathbf{x}_j$$

This is the dot-product between two vectors. The dot-product for standardized vectors happens to coincide with the cosine-similarity that is identical to the correlation-coefficient r (both are measures for similarity or nearness). With the kernel trick, we can now apply many different similarity-measures that hopefully help to distinguish our classes better than euclidean distance can do. Or, put in another way: we can either project our data into higher spaces and compute our conventional similarity measures in those spaces **or**: we use other similarity-measures that approximate the former approach.