svm

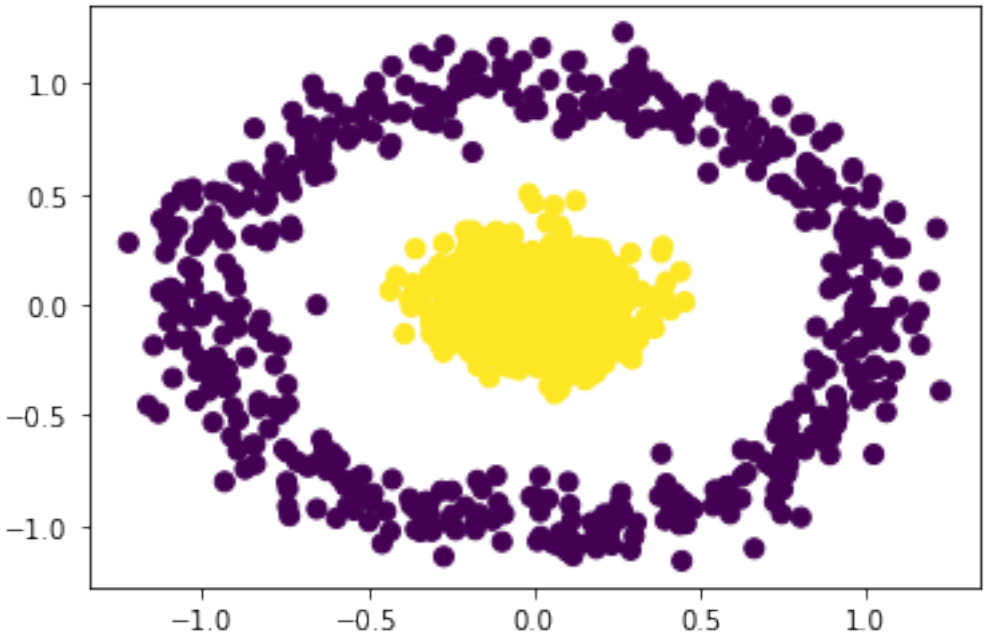
July 6, 2021

1 Support Vector Machines (SVM)

1.1 Beispiel 1: Grundprinzip

Wir erstellen zuerst einen "künstlichen" Datensatz und visualisieren diesen mit *matplotlib*:

[15]:
```
%matplotlib inline
import matplotlib.pyplot as plt
from sklearn.datasets import make_circles

X, y = make_circles(1000, factor=.2, noise=.1)

plt.scatter(X[:, 0], X[:, 1], c=y, s=50)
plt.show()
```



Wir haben also zwei Klassen, wobei die eine Klasse in Kreisform die andere Klasse einschließt. Es gibt hier keine Möglichkeit, diese beiden Klassen mit einer Entscheidungsgrenze voneinander zu

trennen. Deshalb transormieren wir die Daten in den 3-Dimensionalen Raum, hier ganz simpel indem wir jweils die X-Daten quadrieren und die Summe bilden.

Anschließend plotten wir die Daten in einem 3D-Plot. Nun können wir uns eine Ebene vorstellen, die diese beiden Klassen voneinander trennt!

[16]:
```
%matplotlib notebook
from mpl_toolkits import mplot3d
z = (X ** 2).sum(1)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')


ax.scatter(X[:, 0], X[:, 1], z, c=y, s=50)
plt.draw()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Wir versuche nun mit unterschiedlichen Kernels mittels SVM ein Modell zu erstellen. Versuchen wir es zuerst mit dem linearen Kernel!

[3]:
```
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y,shuffle=True,␣
 ↪test_size=0.3, random_state=42)
model_linear = SVC(kernel="linear").fit(X_train, y_train)
print(model_linear.score(X_test, y_test))
```

```
0.47
```

Wie erwartet ist die Accuracy recht schlecht! Versuchen wir es mit einem Polynomialen Kernel mit degree=2:

[4]:
```
model_poly = SVC(kernel="poly", degree=2).fit(X_train, y_train)
print(model_poly.score(X_test, y_test))
```

```
1.0
```

Jetzt haben wir 100% Genauigkeit! Die Klassen können nun also exakt separiert werden. Versuchen wir es aus Spaß noch mit dem rbf-Kernel:

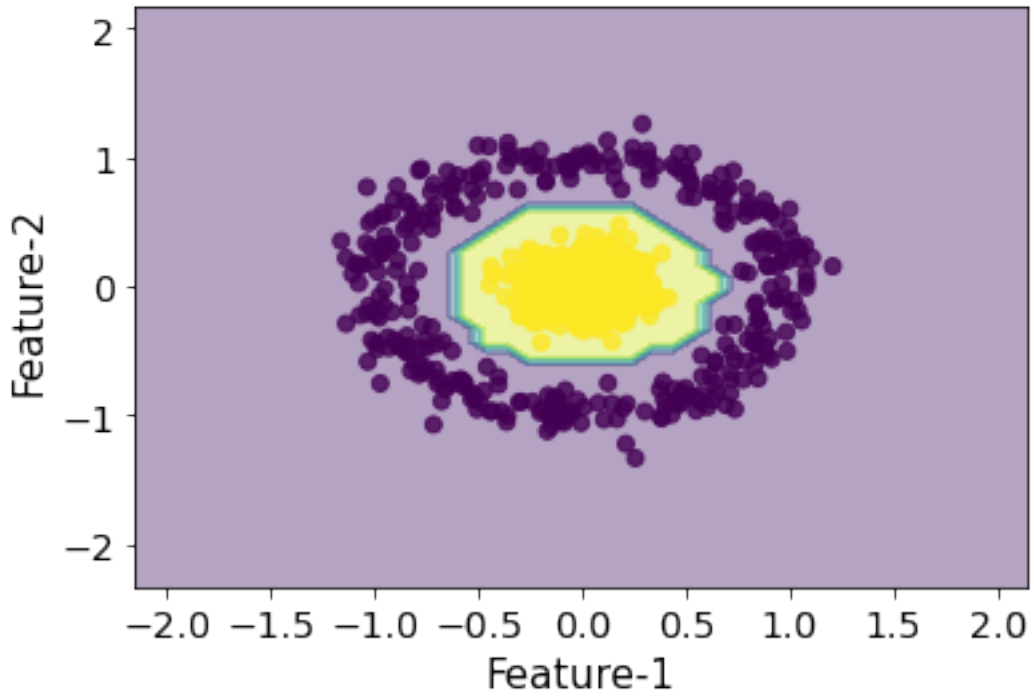
[5]:
```
model_rbf = SVC(kernel="rbf").fit(X_train, y_train)
print(model_rbf.score(X_test, y_test))
```

```
1.0
```

Auch hier 100% Accuracy! Ist auch nicht wirklich überraschend, da sich die Klassen, wie im ersten Plot gezeigt, nicht überlappen und somit mit allen Kernels, die nicht linear sind, gut trennen lassen. Wir können die Entscheidungsgrenze auch mit der Hilfsdatei *plot_decision_boundaries* visualisieren. Diese Datei muss im gleichen Verzeichnis wie diese Jupyter Notebook - Datei liegen!

[6]:
```
%matplotlib inline
from plot_decision_boundaries import plot_decision_boundaries
plot_decision_boundaries(X_train, y_train, SVC, kernel="poly", degree=2)
```

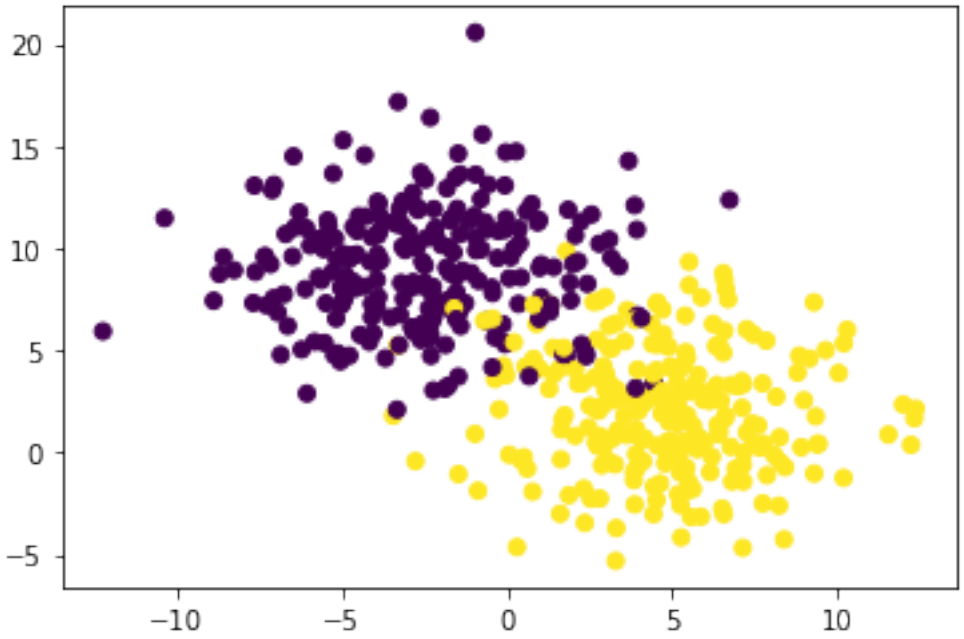
[6]:
```
<module 'matplotlib.pyplot' from 'C:\\Users\\dea40349\\Anaconda3\\lib\\site-
packages\\matplotlib\\pyplot.py'>
```



1.2 Beispiel 2: Entscheidungsgrenzen mit künstlich erzeugten Daten

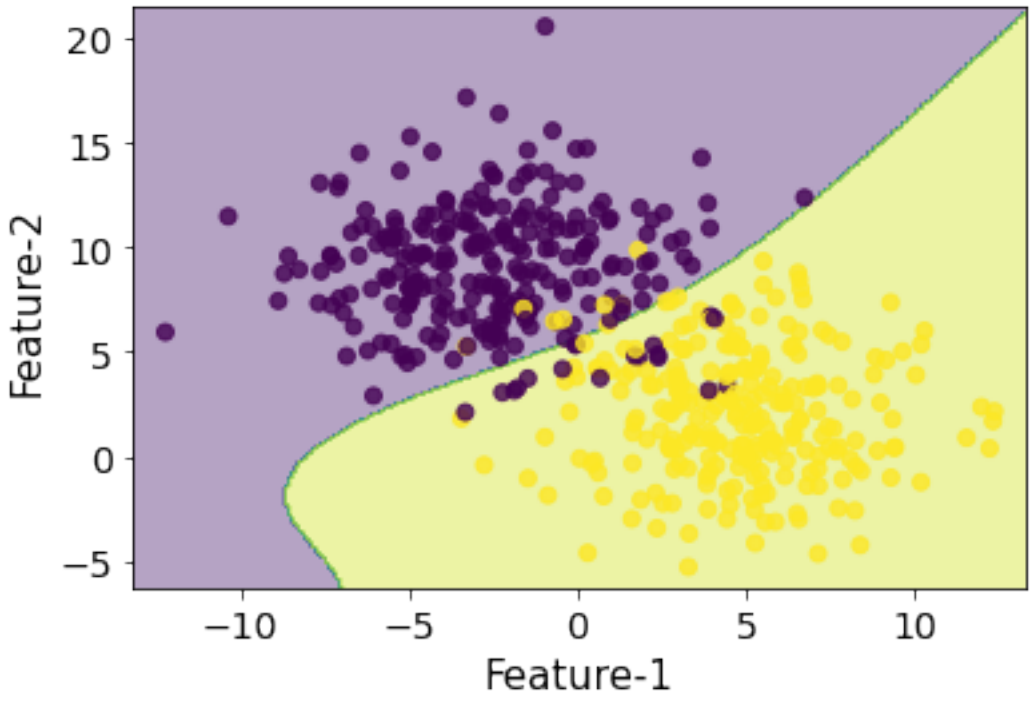
Im 2. Beispiel wollen wir wieder einen künstlich erzeugten Datensatz verwenden. Hier überlappen sich die Objekte der unterschiedlichen Klassen teilweise.

[7]:
```
%matplotlib inline
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

X, y = make_blobs(n_samples=500, centers=2, n_features=2, cluster_std=3,␣
 ↪random_state=42)

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



[8]:
```
%matplotlib inline
plot_decision_boundaries(X, y, SVC, kernel="poly", degree=3)
```

[8]:
```
<module 'matplotlib.pyplot' from 'C:\\Users\\dea40349\\Anaconda3\\lib\\site-
packages\\matplotlib\\pyplot.py'>
```



[9]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.3,␣
↪random_state=42, shuffle=True)
model_lin = SVC(kernel="linear").fit(X_train, y_train)
print(model_lin.score(X_test, y_test))
```

```
0.96
```

[10]:
```
model_poly = SVC(kernel="poly", degree=3).fit(X_train, y_train)
print(model_poly.score(X_test, y_test))
```

```
0.96
```

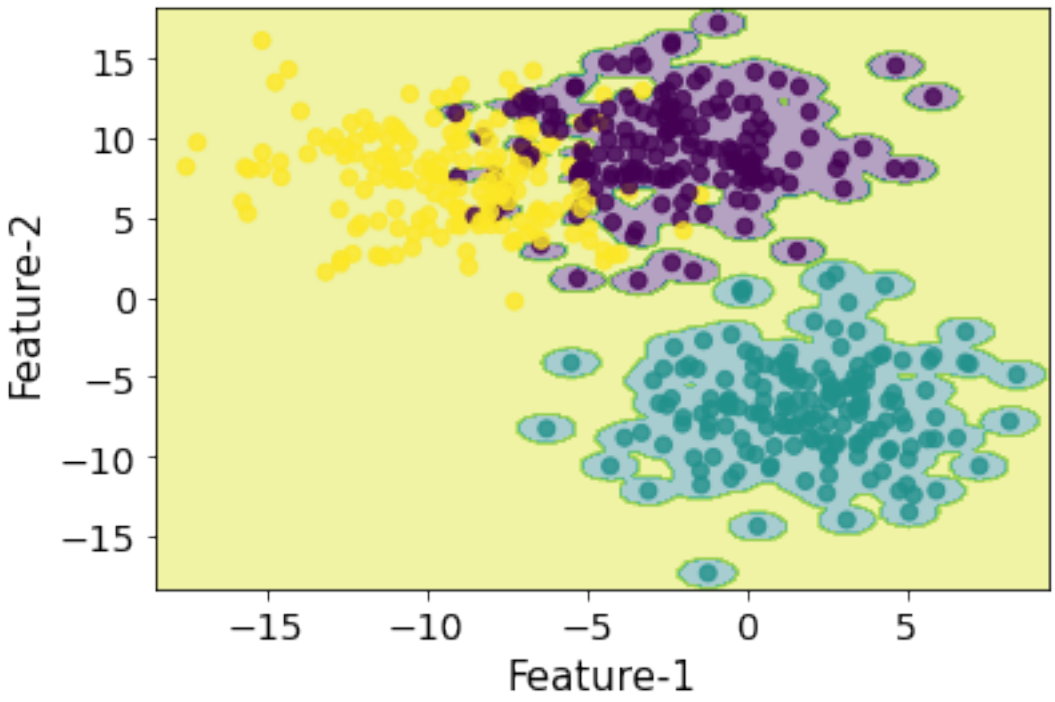
1.3 Hyperparameter

Die wichtigsten:

- **kernel**: 'linear', 'poly', 'rbf', 'sigmoid'
- **poly**: Bei Kernel "poly" der Grad
- **C**: Regularisierungsparameter("Penalty"). Wirkt ggf. Overfitting entgegen. Hoher Wert: "Harte" Grenze, kleiner Wert: "Weiche" Grenze
- **gamma**: {'scale', 'auto'} or float. Wie groß der Einfluss weiter entfernter Punkte ist. Großer Wert: Punkte mit größerem Abstand werden stärker berücksichtigt (näher liegende dafür weniger stark).

[11]:
```
# Klassischer Fall von Overfitting:

X, y = make_blobs(n_samples=500, centers=3, n_features=3, cluster_std=3,␣
↪random_state=42)
plot_decision_boundaries(X, y, SVC, kernel="rbf", C=10, gamma=5)
```

[11]:
```
<module 'matplotlib.pyplot' from 'C:\\Users\\dea40349\\Anaconda3\\lib\\site-
packages\\matplotlib\\pyplot.py'>
```



1.4 Beispiel 3: Ziffernerkennung MNIST-Datensatz

Wir wollen nun mit Hilfe einer SVM handschriftlich geschriebenen Ziffern klassifizieren. Wir verwenden die K-Fold-Cross-Validation im Zusammenhang mit einer Grid-Search, um verschiedene Hyperparameter zu testen. Da die Parameter C und Gamma Fließkommazahlen sind verwenden wir hier die *RandomizedSearchCV*-Klasse.

Vor der Verwendung einer SVM sollte man die Daten standardisieren!

[12]:
```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.preprocessing import StandardScaler

from sklearn.datasets import load_digits
import numpy as np

data = load_digits()
X, y = data.data, data.target

X = StandardScaler().fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
↪random_state=42, shuffle=True)

parameter_grid = {"kernel" : ["linear", "poly", "rbf"],
```

```
                  "degree" : [2,3,5],
                  "C": np.arange(2, 10, 2),
                  "gamma": np.arange(0.1, 2, 0.2)}


grid = RandomizedSearchCV(SVC(), param_distributions = parameter_grid,␣
 ↪n_iter=10, scoring="accuracy",
                           n_jobs=-1, verbose=3, cv=10, random_state=42)
grid.fit(X_train, y_train)
print(grid.best_params_)

model = SVC(kernel="poly", gamma=0.3, degree=3, C=4).fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
Fitting 10 folds for each of 10 candidates, totalling 100 fits
{'kernel': 'poly', 'gamma': 0.30000000000000004, 'degree': 3, 'C': 4}
0.987037037037037
```