Support Vector Machine Kernels

Module 2 | Chapter 4 | Notebook 3

In the last lesson, we only used the SVM's linear kernel. Now we want to go one step further and look at more kernels to unleash the true power of SVMs. In this Lesson you will learn:

- What a kernel is.
- Which principle the SVM follows to generate nonlinear decision boundaries.
- How you use the `'poly'` and `'rbf'` kernels.

The kernel trick

In the last lesson we learned that the kernel determines the learning strategy of an SVM. It's only possible to understand this with the help of mathematical formulas. This would go beyond the scope of this course and would require a deeper knowledge of linear algebra. We will use a heuristic explanation.

Roughly speaking, an SVM tries to solve a set of special equations in parallel. This is called the *quadratic programming problem* (here's a link if you're interested). To solve this efficiently, you often have to transform your data with various methods (e.g. putting it into a space with more dimensions). There are two possibilities for this:

- Approach 1: First you transform your data and then you deal with the quadratic programming problem.
- Approach 2: First you deal with the quadratic programming problem and transform your data in between.

Depending which approach you choose, the transformations differ immensely. If you think long enough about the mathematics behind it, you realize that it's much more efficient to do the transformation in between. Fascinatingly, the reason for this is that transformations that you make in between (approach2) are much much simpler compared to transformations at the beginning (approach 1). These simple transformations are called **kernel functions** and the mathematical insight to take the second approach is called the **kernel trick**. Without question it's this kernel trick that gives an SVM its real power. A linear kernel means that you do nothing with the data for the transformation.

Curved separators - the polynomial kernel

Let's look at an example data set which cannot be separated with a straight decision line. Run the following cell to import *read nonlinear_data.csv* as `df_nonlinear`:

```
In [1]: import pandas as pd
        df_nonlinear = pd.read_csv('nonlinear_data.csv')
        df_nonlinear.head()
```

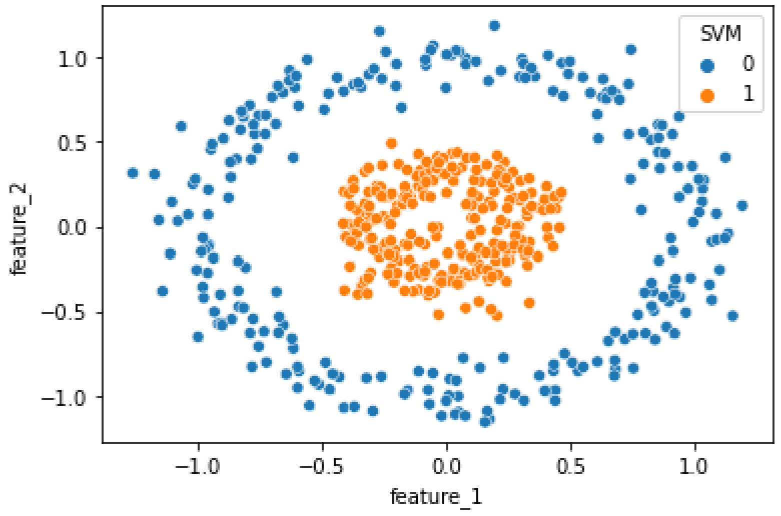
Out[1]:

	feature_1	feature_2	class
0	-0.697055	0.765257	0
1	0.104857	0.048986	1
2	0.254989	0.230096	1
3	0.025126	-0.284946	1
4	0.206967	-0.526438	1

The data consists of two `'feature'` columns and one column with the target vector, `'class'`. Visualize the data with a scatter plot, the color of the data points should depend on the `'class'` column.

```
In [2]: import matplotlib.pyplot as plt
        import seaborn as sns

        fig, ax = plt.subplots( )
        sns.scatterplot(x=df_nonlinear.loc[:,'feature_1'], y=df_nonlinear.loc[:,'feature_2'],
        ax.legend(title='SVM');
```



As you can see, the `0` category forms a circle around the `1` category. We can't separate the categories from one another with a simple straight line. Let's try it anyway and see what result we get. The features are already on the same scale, so we can train the SVM straight away. Run the following cell to do this.

```
In [3]: # import SVC and cross_val_score
        from sklearn.svm import SVC
        from sklearn.model_selection import cross_val_score
```

```
# split df into features and target
features = df_nonlinear.loc[:, ['feature_1', 'feature_2']]
target = df_nonlinear.loc[:, 'class']

# training and validation
model_lin = SVC(kernel='linear')  # instantiate and train linear svm

# 5-fold cross-validation using f1-score and all available cpus
scores = cross_val_score(estimator=model_lin, X=features, y=target, cv=5, scoring='f1'

print('Mean f1-score:', scores.mean())
```

```
Mean f1-score: 0.7244618855586284
```

We get an average F1 score of about 0.72 when we validate this model. This value may not sound too bad to you, but when we draw the decision line, it looks like this:

Decision line is no good here

All the points above the decision line are classified as category `1`. This is represented by the purple color of the area in the background. All the points below the decision line are classified as category `1`. More than half of the points in category `0` were classified incorrectly! So a linear separation is really not suitable for this data and we would say that this model is extremely underfitted (see *Feature Engineering with Polynomials in Module 1, Chapter 4*).

Now let's take a look at what the **polynomial kernel** does. If you instantiate `SVC` (link to documentation) with the `kernel='poly'` parameter, you instruct the SVM to use a polynomial kernel. This opens up new hyperparameters.

- The first is `degree` and indicates the degree of the polynomial. With this information, the SVM creates new polynomial features up to the specified `degree`. We learned how to create polynomial features in *Polynomial Regression (Module 1, Chapter 4)*.
- The second parameter is `gamma`. This parameter controls the importance (the coefficients) of the new polynomial features. A small `gamma` value corresponds to a small influence and a high `gamma` value corresponds to a large influence for the new features. The specific value `gamma='scale'` causes this parameter to be selected automatically.

Since the SVM represents the data in a higher-dimensional space, it will try to find a decision hyperplane with the largest margin in thisspace, just as it does with `kernel='linear'`. Due to the increased dimension, this is often successful.

For our data, the result would look like this:

Paraboloid

We see that our data is now also presented in three dimensions and that the separation is now from top to bottom.

Now initialize `SVC` from the `sklearn` module `sklearn.svm` with the parameters `kernel='poly'`, `degree=2` and `gamma='scale'`. Assign the initialization to the variable `model_poly`.

In [4]:
```
model_poly = SVC(kernel='poly', degree=2,gamma='scale')
```

Now use `cross_val_score()` to evaluate the model, similarly to the code cell above. Use 5-fold cross validation and the F1 score as the quality metric. If you want to speed up the calculation, you can still use `n_jobs=-1` (then use all CPU cores for the calculation). Assign the result to the variable `scores_poly` and then print the mean value of the obtained scores.

In [5]:
```
scores = cross_val_score(estimator=model_poly, X=features, y=target, cv=5, scoring='f1
print('Mean f1-score:', scores.mean())
```

```
Mean f1-score: 1.0
```

With an F1 score of 1.0, we were able to classify all the data points in the validation set correctly with this kernel. So the polynomial kernel seems very well suited for this data. The decision plane now looks like this:

Categories are separated here

Note that in this image - compared to the three-dimensional image above - all the data points were projected onto the decision plane and we're representing the decision plane itself.

Congratulations: Now you have an idea of how a nonlinear SVM works and when a polynomial kernel can help.

The Gaussian kernel

Probably the most commonly used nonlinear kernel is the `'rbf'` kernel. This is also called the **Gaussian kernel** and is the default setting for the `kernel` parameter. The abbreviation stands for *Radial Basis Function*. The corresponding transformation of the data is as follows:

- All the data points are made to "run" like drops of ink. Imagine that our scatter plot above showed drops of ink instead of data points. These drops of ink run, leaving most ink in the middle of the droplet and less at the edges.
- Then it's measured how much the droplets touch each other (think of two droplets next to each other, where eventually their edges touch). The intensity of this touch point is then a measure for the different categories of data. All possible touch points then form new features.

The number of features that are created is then very large, because we have to calculate the touch point from every point to every other point in the data set. To be more precise, the following formula applies:

\begin{equation}\textbf{Number of generated features} = \left(\left(\textbf{Number of data points}\right)\cdot\left(\textbf{Number of data points} - 1\right)\right)/2\end{equation}
For example, we get `45` features for `10` data points and `4950` features for `100` data points. So the number of features (and therefore dimension) increases very quickly. Although this can

lead to long calculation times, it helps the SVM tremendously to find a separation hyperplane with the widest *margin*.

Remember that SVMs always look for a separation hyperplane. However, this is often only possible in higher-dimensional spaces. This is also the reason why SVMs take a very long time to train and are work best with medium and small data sets.

How well does this kernel cope with our example data? Define `model_rbf` as an instance of `SVC` with the parameters `kernel='rbf'` and `gamma='scale'`. The `degree` parameter is only used for the polynomial kernel. So you don't need to specify it here. Then use `cross_val_score()` to evaluate the model and store the result in the variable `'scores_rbf'`. Then print the average F1 score as before.

In [6]:
```
model_rbf = SVC(kernel='rbf' , gamma='scale')
scores = cross_val_score(estimator=model_rbf, X=features, y=target, cv=5, scoring='f1'
print('Mean f1-score:', scores.mean())
```

```
Mean f1-score: 1.0
```

Once again, we get the ideal value of 1.0. So this kernel can handle our data very well! In this example, the decision limits are as follows:

Decision line doesn't fit here

The result is very similar to the polynomial case, only the radius of the circle is slightly smaller. But it still includes all the same points as the polynomial kernel.

Congratulations: Now you've also got to know the `'rbf'` kernel. It achieves good results with a lot of data sets, which is why it is so frequently used.

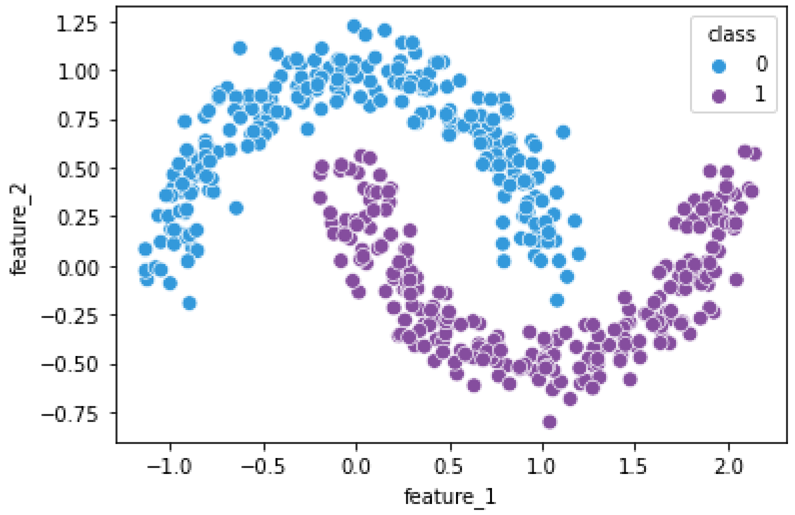
Kernel adaptability

You've seen that the `'rbf'` kernel as well as `'poly'` with `degree=2` are both able to separate our example very effectively. They both create very similar decision boundaries. This could lead to the impression that they're not particularly different from one another. Is this the case?

Let's try it out. Run the following cell to import and visualize another two-dimensional data set.

In [7]:
```
df_nonlinear_other = pd.read_csv('nonlinear_data_2.csv')

sns.scatterplot(data=df_nonlinear_other, x='feature_1',
                y='feature_2', hue='class',
                palette=['#3399db', '#854d9e'], s=60);
```



Now the data points are now no longer arranged in a circle, but in two interlocking arcs. You can see straight away that you cannot simply separate the categories with a straight line. So the linear kernel will not give us the best results here. But what about the polynomial kernel and the `'rbf'` kernel?

Run the following code cell to calculate the distances to the cluster centers:

In [8]:
```
features_other = df_nonlinear_other.loc[:, ['feature_1', 'feature_2']]
target_other = df_nonlinear_other.loc[:, 'class']
```

Now use `cross_val_score` to evaluate the models `model_poly` and `model_rbf` with this data and store the results in the corresponding variables `scores_poly` and `scores_rbf`. Use a 5-fold cross validation. Then print the average F1 values.

In [9]:
```
model_poly = SVC(kernel='poly', degree=2,gamma='scale')
scores = cross_val_score(estimator=model_poly, X=features_other, y=target_other, cv=5,
print('Mean f1-score poly 2:', scores.mean())
model_poly = SVC(kernel='poly', degree=3,gamma='scale')
scores = cross_val_score(estimator=model_poly, X=features_other, y=target_other, cv=5,
print('Mean f1-score poly 3:', scores.mean())
model_poly = SVC(kernel='poly', degree=4,gamma='scale')
scores = cross_val_score(estimator=model_poly, X=features_other, y=target_other, cv=5,
print('Mean f1-score poly 4:', scores.mean())
model_poly = SVC(kernel='poly', degree=5,gamma='scale')
scores = cross_val_score(estimator=model_poly, X=features_other, y=target_other, cv=5,
print('Mean f1-score poly 5:', scores.mean())
scores = cross_val_score(estimator=model_rbf , X=features_other, y=target_other, cv=5,
print('Mean f1-score rbf:', scores.mean())
```

```
Mean f1-score poly 2: 0.8309682338318021
Mean f1-score poly 3: 0.9376329827401845
Mean f1-score poly 4: 0.8306461898609475
Mean f1-score poly 5: 0.9346955225459899
Mean f1-score rbf: 0.998019801980198
```

We get an F1 score of about 0.83 for `kerel='poly'` with `degree=2`. For `kernel='rbf'` we get an almost perfect score of about 0.998. The `'rbf'` kernel seems to be able to adapt better to this data. This will become clearer if we visualize the decision boundaries:

Decision line doesn't fit here

On the left is the polynomial kernel. This is only able to adapt to the data to a limited extent. However, the decision boundary of `'rbf'` looks much more flexible. It manages to separate the categories quite well. This is achieved by creating a large number of new dimensions based on the distances between the points. Thanks to the flexibility of its decision boundary, this kernel often achieves good results with relatively little effort.

Despite the kernel trick, nonlinear kernels require more computing power. So we recommend trying out the linear kernel first. SVMs generate predictions relatively quickly, but are quite slow to train. If it is critical that you can train your SVM quickly, you can use `LinearSVC` from `sklearn.svm`. This is a different implementation of the linear SVM - (documentation link). It scales better with a lot of data points.

Congratulations: Now you know two non-linear kernels. You can use them to set up your SVM flexibly. With this you have added a lot more classifiers to your tool kit.

Remember:

- Use polynomial kernels with `kernel='poly'`, `degree` and `gamma`.
- Use the gaussian kernel with `kernel='rbf'` and `gamma`.
- SVMs always form decision lines, planes or hyperplanes. The kernels then decide which dimensions to work in and with which features.

Do you have any questions about this exercise? Look in the forum to see if they have already been discussed.
