Support Vector Machines

Module 2 | Chapter 4 | Notebook 2

In this chapter we will get to know a new, powerful machine learning algorithm: the Support Vector Machine (SVM for short) It is widely used and often achieves good results. In this notebook you will learn:

- What the support vectors are
- How SVMs use them to make predictions
- How to interpret the `C` parameter of the SVM

Support Vectors

Historically, the *support vector machine* (SVM) is a very young algorithm, which only gained relevance in the mid-90s. This makes it all the more impressive that it is one of the most popular and flexible algorithms ever. SVMs can perform both classifications and regressions and they are often unbeatable in their prediction power for medium and small size data sets. Without question, every data scientist should have them in their repertoire.

In this chapter you'll get to know SVMs as a classifier. But before we jump into a scenario, let's start with a small example to understand how SVMs work.

Import the file `sample_data_svm.csv` and save the data as a `DataFrame` named `df`. Then print the first five rows.

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

Out[1]:

	feature_1	feature_2	class
0	7.936991	-2.794593	0
1	0.684791	-7.217534	1
2	7.340057	-3.737374	0
3	2.337024	-8.301279	1
4	0.852193	-4.945512	1

As you can see, the data has two `'feature'` columns and one `'class'` column. The latter describes which category the data point belongs to. Our goal is to use SVMs in a way that allows us to make predictions about `'class'` membership.

`sklearn` offers us different implementations of the SVM. They are all located in the `sklearn.svm` module. For classification problems we use the *support vector classifier*, `SVC` for short (link to documentation). The most important hyperparameters for `SVC` are:

```
SVC(C=float,      # regularization parameter, controls the 'strictness' of
the SVM
    kernel=str # kernel parameter, controls the learning style of the SVM
   )
```

The choice of `kernel` is particularly important. This parameter tells the SVM which learning strategy to follow. So `kernel` directly influences the predictive power and efficiency of the SVM.

In the simplest case, the SVM uses a linear function for its calculations. In the next lesson we'll learn about other kernels and their effects.

Now import `SVC` from `sklearn.svm` and instantiate it as `model_svm`. Use the parameters `kernel='linear'` and `C=10`. You'll see what `C` does in the second part of this lesson.

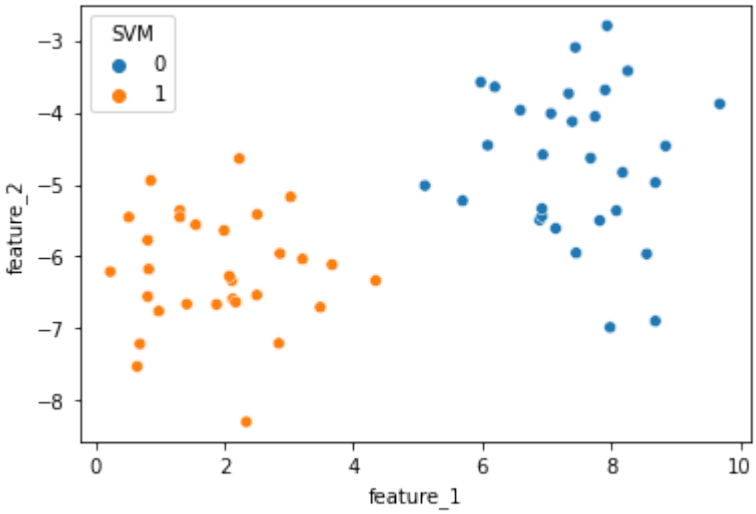
In [2]:
```
from sklearn.svm import SVC
model_svm = SVC(kernel='linear', C=10)
```

To better understand how the SVM works, let's take a look at the data. Draw a scatter plot of data with `'feature_1'` on the x axis and `feature_2'` on the y axis. Color the data points according to the `'class'` column.

In [3]:
```
import matplotlib.pyplot as plt
import seaborn as sns

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

Out[3]:
```
<matplotlib.legend.Legend at 0x7f775569e610>
```



The SVM now wants to separate the categories with a straight line (because `kernel = 'linear')`. There are different possibilities. The following illustration shows two different

decision lines, which can separate the categories beautifully:


Possible decision lines

As soon as the SVM finds a line like this , the prediction is very easy - the side a data point is located on determines which class it belongs to. However, you can see that both decision lines are very different. The vertical line is reminiscent of the decision tree, which only decision lines that are parallel to the axes, i.e. based on only one feature (in this case it would be `'feature_1'`). In contrast, the diagonal line uses both features to separate the data points.

But which is the best decision line? The SVM can answer this question. This is because it searches for the "widest gap" between the data categories. In this context, "broad" means the most extensive version of the decision line. Imagine widening the decision line until it touches one (or more) data points.

In the case above, the SVM would thus choose the diagonal decision line. The following figure illustrates this result:


SVM decision line

The outer lines are always equidistant from the decision line and they touch one or more data points (three in the case above).

Even if things have just become a little abstract, we should always keep in mind that the points of contact are still individual data points from our data set. The combination of the feature values of the points touching the margins define the ***support vectors*** because they completely determine both the decision line and the width of the gap, i.e. they **support** the decision line. We don't have time or space in this course to go into proving this fact mathematically. In the context of SVMs, the perpendicular distance between any support vector and the decision line is called the ***margin*** (so the width of the gap is twice the margin).

At this point it becomes clear how the SVM works: The algorithm searches for the **decision line with the largest perpendicular distance to the *support vectors* (*margin*)**.

`SVC` stores both the *support vectors* and a way to find the decision line. But before we can accessit, we first have to train our model.

Attention: The distance between the data points plays an important role. It is largely determined by the scales of the features. So when using SVMs, it's important to put all the features on the same scale. In our case you can use `StandardScaler()` from `sklearn.preprocessing`.

Import `StandardScaler` and instantiate the scaler as `scaler`. Then create the new variable `features`, which consists of the standardized values of the columns `['feature_1', 'feature_2']`. Then fit `model_svm` to `features` and use the `'class'` column as the target vector.

In [4]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

features =scaler.fit_transform(df.loc[:, ['feature_1', 'feature_2']])
target = df.loc[:, 'class']
model_svm.fit(features, target)
```

Out[4]:
```
SVC(C=10, kernel='linear')
```

The support vectors (the data points touching the "gap") are now located in the attribute `my_model.support_vectors_`. Furthermore, `SVC` stores the row numbers of the support vectors in the training set in the `my_model.support_` attribute. Check that the rows in `model_svm.support_vectors_` match the rows in `df` indicated by `model_svm.support_`. Remember only to use the features for comparison.

In [5]:
```
print(model_svm.support_vectors_)

print(model_svm.support_)

features[model_svm.support_, :]
```

Out[5]:
```
[[ 0.14752874  0.36479281]
 [-0.55998514  0.23287603]
 [-0.11143339 -0.74587857]]
[34 43 50]
array([[ 0.14752874,  0.36479281],
       [-0.55998514,  0.23287603],
       [-0.11143339, -0.74587857]])
```

We get the value `True` six times because we have 3 support vectors with 2 features each. Now for the decision line: It is defined by the attribute `my_model.coef_`. It contains a vector which is perpendicular to the decision line. Using vector analysis tools, you can then calculate the decision line.

Now print `model_svm.coef_`.

In [6]:
```
model_svm.coef_
```

Out[6]:
```
array([[-2.60440205, -1.1927926 ]])
```

We get numerical values of about -2.60 for `'feature_1'` (first position of the vector) and -1.19 for `'feature_2'` (second position of the vector). These values may differ for you, but the ratio should be the same. The individual numbers of the vector quantify the importance of the features for the SVM. In our case, `'feature_1'` has a higher absolute value and is therefore marked as more important. This is confirmed by the visual impression: The categories differ more on the x-axis than on the y-axis. If the SVM had opted for the vertical decision line, the other value of `feature_2` would be 0. Because this feature would not come into play.

Congratulations: You've trained your first SVM! Now you known that the *support vector machine* tries to create a decision line between the data. It uses the *support vectors* to find the "widest gap" between the categories.

SVM flexibility: *hard margin* and *soft margin*

So the SVM tries to separate the categories with a line and leave as much buffer as possible between the categories. But it's rarely possible to separate the data as nicely as in the previous example. To make this clear, we'll add a new point to the data which will greatly reduce the *margin*. Add the point with the features `[4.1, -5.7]` and the class `0` to `df`. Then visualize the data again as a scatterplot.

Tip: `df` has 60 data points so far.

In [7]:
```
new_row = {"feature_1": 4.1, "feature_2": -5.7, 'class': 0}
df.append(new_row, ignore_index=True)
df
```

Out[7]:

	feature_1	feature_2	class
0	7.936991	-2.794593	0
1	0.684791	-7.217534	1
2	7.340057	-3.737374	0
3	2.337024	-8.301279	1
4	0.852193	-4.945512	1
5	0.807189	-6.559665	1
6	6.591051	-3.968351	0
7	0.225655	-6.213492	1
8	0.820561	-6.178497	1
9	6.890990	-5.494420	0
10	1.551463	-5.561054	1
11	2.840811	-7.209978	1
12	8.686735	-4.974945	0
13	7.066294	-4.015369	0
14	5.694223	-5.227158	0
15	8.552577	-5.967043	0
16	8.082632	-5.365122	0
17	3.668537	-6.114906	1
18	0.806690	-5.774963	1
19	0.641881	-7.530546	1
20	7.750573	-4.055847	0
21	2.499529	-6.539526	1
22	2.124881	-6.591610	1
23	7.822596	-5.502551	0
24	6.928525	-5.440311	0
25	2.173277	-6.636042	1
26	1.412777	-6.663294	1
27	1.305534	-5.362719	1
28	7.906773	-3.687573	0
29	2.505696	-5.418793	1
30	0.977683	-6.762671	1
31	7.681613	-4.635222	0
32	6.084064	-4.456504	0
33	7.396241	-4.126682	0

	feature_1	feature_2	class
34	5.109826	-5.016994	0
35	7.983571	-6.987507	0
36	6.939833	-4.586733	0
37	3.208446	-6.037237	1
38	8.258496	-3.422992	0
39	8.686234	-6.900693	0
40	8.845677	-4.466223	0
41	1.992502	-5.641453	1
42	3.484748	-6.707557	1
43	3.024451	-5.173822	1
44	2.859952	-5.961955	1
45	6.195084	-3.644549	0
46	8.177905	-4.833472	0
47	7.444641	-3.095886	0
48	1.305277	-5.454193	1
49	5.976153	-3.578601	0
50	4.346543	-6.337408	1
51	7.456802	-5.953018	0
52	6.927286	-5.337504	0
53	9.686491	-3.880111	0
54	0.512341	-5.455132	1
55	2.113829	-6.337482	1
56	1.874704	-6.669129	1
57	2.227087	-4.641987	1
58	2.074021	-6.276236	1
59	7.139341	-5.611503	0

As you can see, the new data point is closer to class `1` than to class `0`. This could be an outlier, for example. We quickly realize that it is very difficult to position a straight "wide gap" through the data without any points in it. At first sight, you might wonder whether it makes sense to apply a SVM at all in this case. The answer is a very clear yes (otherwise it would hardly be one of the most popular algorithms).

Now let's look at how the SVM behaves here. Since we've now added a new data point, we have to recreate the `features` variable. Run the following code cell to do this:

In [8]:
```
features = scaler.fit_transform(df.loc[:, ['feature_1', 'feature_2']])  # fit scaler t
```

Now the parameter `C` comes into play. The value of `C` gives the SVM the possibility to "allow individual points into the street". Basically, this means that the SVM now looks for the largest *margin* to the second closest (or third, fourth, fifth, etc.) *support vectors*. There are two different kinds of case here: We say the SVM has a *hard margin* when it doesn't let any points into the "gap", and we call it a *soft margin* when they are allowed. Sometimes we say that `C` controls the **penalty level for incorrect classifications**.

Remember: Small `C` values result in more points in the "gap" (margin) and high `C` values result in fewer points.

Instantiate the `SVC` again, this time using the very high value `C=1000` and use `kernel='linear'` again. Name it `model_svm` again and fit it to the new data.

In [9]:
```
model_svm = SVC(kernel='linear', C=1000)
model_svm.fit(features, df.loc[:, 'class'])
```

Out[9]:
```
SVC(C=1000, kernel='linear')
```

Now the distance between the decision line and support vectors is very small, but there are no points in the "gap". You can see this in the following visualization:

 Hard margin

The new data point dominates the SVM's predictions here, since the margin is measured based on the nearest *support vectors*. This can result in the model predicting new data points very poorly. So in this case it suffers badly from overfitting. So it can be useful to accept incorrect classification when training. If we had used the value `C=0.5`, it would have resulted in the following decision line:

 Soft margin

You can find the best `C` parameter for your own data with a grid search (see *Grid Search (Chapter 1, Module 1)*).

At this point, we should note that data very rarely behaves like it does in our example data. For one thing, you often have a lot more features. This means that the data points (and therefore the *support vectors*) live in higher dimensions, and a decision line becomes either a decision plane (in three dimensions) or a decision hyperplane (in more than three dimensions).

Furthermore, real data often contains errors, so that the categories are even more mixed up. Nevertheless, the SVM will take a decision plane and try to allow as few points into the "gap" as possible (i.e. to classify some points incorrectly).

When classifying data points, the SVM just sees which side of the decision line a data point is located. This has some advantages and disadvantages.

A very direct disadvantage is that the SVM takes a very long time to train (depending on the `kernel`). It's rarely used on large data sets for this reason. However, for small or medium-sized data sets, you should immediately think of the SVM.

Another drawback is that it doesn't return probabilities like logistic regression or random forests, so by default `SVC` does not have the `my_model.predict_proba()` method. However, you can set the `probability=True` parameter can be set during instantiation. The SVM then tries to learn probabilities, using cross validation internally. This makes predicting probabilities very demanding in terms of computing power, especially with large data sets.

One advantage of the SVM method is that it can be done quite quickly, since only very few data points really contribute to the model. This makes the SVM very useful for high-dimensional data. Typically, the distances between data points increase when they have more dimensions. This makes it easier to find a plane that separates the categories. You may hear quite often that SVMs work well for sparse matrices. These are feature matrices that contain a lot of columns and rows, but many of the values are 0. This means that the data points have many features, i.e., dimensions. As just explained, an SVM is often a good choice here. Text data in particular falls into this category. Creating usable features from it part of what's called *Natural Language Processing* (NLP for short). We'll take a closer look at how this works in the next lesson.

Congratulations: Now you know how the outliers can influence a support vector machine's decision plane. You can reduce this influence with the `C` parameter.

Remember:

- Support vector machines are ideal for small or medium data sets.
- The support vector machine creates a decision plane that is supported by the support vectors.
- A high value for `C` results in a *hard margin*, and a low value in a *soft margin*.

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