Support Vector Machine Recap

Module 2 | Chapter 5 | Notebook 1

In this notebook we'll look at the most important things from the previous chapter. This is all repetition. If you discover a lot of new things here, we recommend that you take a look back at Chapter 4. The relevant lessons for each section are clearly marked.

Linear Support Vector Machines

In the first exercise of Chapter 4 *Linear Support Vector Machines*, you learned that SVMs are classifiers, which divide data into different categories with a dividing line or plane.

Let's take a look at the principle of SVMs again and import the sample data.

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 only 2 features and the `'class'` column, which describes which category each datapoint belongs to.

They are located in the `sklearn.svm` module. We use it with the parameters `kernel='linear'` and `C=10` to calculate a decision line between the classes in `'class'`.

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

The SVM's actual decision line is shown here as a solid orange line. The lines parallel to it go through the points closest to the decision line, which have an orange edge The SVM tries to find the widest path between the data points. The width is defined by the distance between the two dashed lines, which is called the margin. The decision line only depends on the marked border

points. None of the other points are relevant for the calculation. This is where the model gets its name. The boundary points support the decision line and are therefore called *support vectors*.

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`.

In [3]:
```
from sklearn.preprocessing import StandardScaler

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

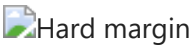
Out[3]:
```
SVC(C=10, kernel='linear')
```

After the model training, the decision plane is determined in the attribute `my_model.coef_`. It contains a vector which is perpendicular to the Direction of the decision plane. We get the vectors with `model_svm.coef_`.

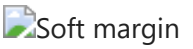
In [4]:
```
model_svm.coef_
```

Out[4]:
```
array([[-2.60440205, -1.1927926 ]])
```

If you don't want the *Support Vector Machine* to misclassify points during training under any circumstances, then we call this hard limit a *hard margin*. The SVM's `C` parameter controls this. In contrast to `LogisticRegression`, `C` here doesn't control the number of features that are used. Instead, it controls whether it is more important to us to have a wide *margin* (low `C` value) or to separate the categories as well as possible (high `C` value). `C` therefore indicates how much misclassifications are penalized. It can be very useful to reduce the `C` value, especially when dealing with outliers. Here is the decision line after adding in an outlier.


Hard margin

The outlier here dominates the SVM's predictions. 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 classifications when training. We call this a *soft margin*. You can achieve this behavior with a low `C` value. If we had used the value `C=0.5`, it would have resulted in the following decision line:


Soft margin

Congratulations: Now you have recapped Support Vector Machines, their Support Vectors, and hard and soft margins.

Nonlinear SVMs

A lot of the time, you can't separate your data in a linear way. The *Support Vector Machine* can then create new dimensions, so to speak, by applying a kernel function, so that it can still find a decision plane. But now it is very demanding in terms of computing power to create a lot of new features with large amounts of data. So you don't really want to do that. This is where you can use a mathematical trick: what's known as the kernel trick.

It involves transforming the original problem of finding a decision plane in data with new dimensions into an equivalent problem where the new dimensions do not have to be created. Instead, the scalar products of the data are calculated. There are functions, called kernels, which can calculate the scalar product of transformed data without performing the actual transformation. Calculating the scalar product requires less computing effort than performing the transformation. Transformation the problem like this makes it possible to obtain the same results without actually transforming the data, saving computing time.

The following example shows data that is not suitable for an SVM with a linear kernel:

In [5]:
```
# import pandas and read data
import pandas as pd
df_nonlinear = pd.read_csv('nonlinear_data.csv')

# import cross_val_score
from sklearn.model_selection import  cross_val_score

features = df_nonlinear.loc[:, ['feature_1', 'feature_2']]  # define features
target = df_nonlinear.loc[:, 'class']  # define target
model_lin = SVC(kernel='linear')  # instantiate and linear svm
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
```

 Decision line is no good here

As you can see, the data points cannot be neatly separated by a straight line. If we use a linear SVM we get a f1 score of 0.72, which is not very good for data points that are this clearly separated.

What might help here is a polynomial kernel with degree 2. It arranges the data on a 3-dimensional paraboloid, so to speak. The new 3rd dimension makes it possible to determine a plane that separates the data very well. We have already seen this in *Chapter 4, Using SVM with Kernels*.

We'll instantiate `SVC` with the parameters `kernel='poly'` and `degree=2` for a polynomial kernel of degree 2 and receive a round separation of the data classes and an ideal f1 score of 1.0. The `gamma` parameter represents the expression of the nonlinear function. A small `gamma` value corresponds to a low level of expression. In our example this means that the points are not lifted so high into the 3rd dimension. This means that they are closer together in this direction, which makes it harder to separate them here. `gamma='scale'` causes this parameter to be selected automatically and is usually a good start.

In [6]:
```
model_poly = SVC(kernel='poly', degree=2, gamma='scale')  #instantiate svm with polync
scores_poly = cross_val_score(estimator=model_poly, X=features, y=target, cv=5, scorir
print('Mean f1-score:', scores_poly.mean())
```

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

We get an equally good separation for this data with the `'rbf'` kernel. The `'rbf'` kernel is probably the most commonly used non-linear kernel and is also the default setting for `SVC`. In the context of Support Vector Machines, this is sometimes called the gaussian kernel. This is because the function is usually implemented in such a way that the values are calculated according to the function of a bell curve.

We'll 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. Then we'll use `cross_val_score()` to evaluate the model.

In [7]:
```
model_rbf = SVC(kernel='rbf', gamma='scale')  # instantiate svm with radial basis func
scores_rbf = cross_val_score(estimator=model_rbf, X=features, y=target, cv=5, scoring=
print('Mean f1-score:', scores_rbf.mean())
```

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

The decision boundary is round for both kernels and looks something like this:


Decision line

The `'rbf'` kernel usually produces good results quickly. It can be applied to very different data sets, but is more complex than the polynomial kernel

Congratulations: You have recapped why we use nonlinear kernels in SVMs.

Natural Language Processing

Natural Language Processing, NLP for short, is a branch of computer science and artificial intelligence, which deals with the interaction between computers and human (natural) languages. In the last chapter, you learned about a popular NLP business application: text tagging (also known as text categorization). The goal of text classification models is to automatically assign texts to one or more categories. We used the data from `text_messages.csv` for this.

In [8]:
```
import pandas as pd
df = pd.read_csv('text_messages.csv', index_col=0)
df.head()
```

Out[8]:

	status	msg
0	0	Go until jurong point, crazy.. Available only ...
1	0	Ok lar... Joking wif u oni...
2	1	Free entry in 2 a wkly comp to win FA Cup fina...
3	0	U dun say so early hor... U c already then say...
4	0	Nah I don't think he goes to usf, he lives aro...

NLP can be used to automatically classify these text messages as either normal or spam. First of all, the text data is cleaned up to correct errors and to give it more structure for analysis (Preparing and Cleaning Data). Then NLP methods are used to obtain certain parts of the text which reflect the categories as well as possible. These extracted characteristics can then be used with the classification models you have already got to know.

We'll need the modules `string`, `re` as well as `spacy` and `nltk` to clean and prepare the data.

`spacy` and `nltk` offer similar tools for NLP, but have they different approaches. Functions from `nltk` accept `str` values and output `str` values. `spacy` uses an object-oriented approach and generally returns document objects, which have their own attributes and methods. Many users find `spacy` to be more time and memory efficient than `nltk` and therefore more suitable for production.

To use `spacy`, we load the corresponding statistical model for English `en_core_web_sm` with the function `spacy.load()`.

In [9]:
```
# import modules for nlp
import string, re, spacy, nltk

# load language model
nlp = spacy.load('en_core_web_sm')
```

```
2024-04-26 09:49:23.867847: W tensorflow/stream_executor/platform/default/dso_loader.
cc:60] Could not load dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.
0: cannot open shared object file: No such file or directory
2024-04-26 09:49:23.867885: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Igno
re above cudart dlerror if you do not have a GPU set up on your machine.
```

In order to analyze text data correctly, machine learning models should be able to recognize structures in the text, such as individual words and their language components. Therefore, we first have to tokenize the text using a `Doc` object, which means dividing the corpus up into meaningful linguistic units, such as words or sentences. These are then called *token*. A `.Doc` object is a sequence of `token` objects, which are the individual linguistic units we need for our analysis. The text of a `token` is in the attribute `my_token.text`.

In [10]:
```
doc = nlp(df.loc[105, "msg"])  # take example message
doc_tokens = [token.text for token in doc]  # split message into tokens and take the t
print(doc_tokens)
```

```
['Thanks', 'a', 'lot', 'for', 'your', 'wishes', 'on', 'my', 'birthday', '.', 'Thank
s', 'you', 'for', 'making', 'my', 'birthday', 'truly', 'memorable', '.']
```

`spacy` not only divides the message up, but also performs *part of speech tagging* (POS tagging) This means that for each `token`, the type of word is indicated, such as nouns (`'NOUN'`) or punctuation (`'PUNCT'`). We can access this with the `my_token.pos_` attribute.

In [11]:
```
token_pos = [[token.text, token.pos_] for token in doc]  # get parts of speech tags
print(token_pos)
```

```
[['Thanks', 'NOUN'], ['a', 'DET'], ['lot', 'NOUN'], ['for', 'ADP'], ['your', 'PRON'],
['wishes', 'NOUN'], ['on', 'ADP'], ['my', 'PRON'], ['birthday', 'NOUN'], ['.', 'PUNC
T'], ['Thanks', 'NOUN'], ['you', 'PRON'], ['for', 'ADP'], ['making', 'VERB'], ['my',
'PRON'], ['birthday', 'NOUN'], ['truly', 'ADV'], ['memorable', 'ADJ'], ['.', 'PUNC
T']]
```

Typical cleaning and preparation techniques include: Lemmatization, removing stop words and removing punctuation marks. With the `my_token.lemma_` attribute, you can obtain the root form of the corresponding `token`. POS tagging is used to determine the appropriate lemma. This process is called lemmatization.

Pronouns are tagged in `spacy` as `'-PRON-'`. We can remove these with a list comprehension, for example:

In [12]:
```
doc = [token.lemma_ for token in doc if token.pos_ != "PRON"]  # ignore pronouns
print(doc)
```

```
['thank', 'a', 'lot', 'for', 'wish', 'on', 'birthday', '.', 'thank', 'for', 'make',
'birthday', 'truly', 'memorable', '.']
```

Removing stop words means taking out words that appear very frequently. These generally provide very little information about the text and dilute it, so to speak. You can find a list of stop words in `nltk.corpus`. We'll convert them into a `set`, since this is optimized for when we want to search for elements in it. We can also remove the stop words with with a list comprehension.

In [13]:
```
# get stopwords
from nltk.corpus import stopwords
stopWords = set(stopwords.words('english'))

print("LEMMATIZED TEXT:\n", doc)
doc = [token.lower() for token in doc if token not in stopWords]  # ignore stopwords
print("NO STOP WORDS:\n", doc)
```

```
LEMMATIZED TEXT:
 ['thank', 'a', 'lot', 'for', 'wish', 'on', 'birthday', '.', 'thank', 'for', 'make',
'birthday', 'truly', 'memorable', '.']
NO STOP WORDS:
 ['thank', 'lot', 'wish', 'birthday', '.', 'thank', 'make', 'birthday', 'truly', 'mem
orable', '.']
```

Then we can remove the punctuation marks, as they dilute the text a bit like stop words.

In `string.punctuation` there is a *string* with punctuation marks and symbols. We can also a list comprehension to remove the punctuation.

In [14]:
```
punctuations = string.punctuation
print('Satzzeichen:', punctuations)

doc = [token for token in doc if token not in punctuations]  # ignore punctuations
print("NO PUNCTUATIONS:\n", doc)
```

```
Satzzeichen: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
NO PUNCTUATIONS:
 ['thank', 'lot', 'wish', 'birthday', 'thank', 'make', 'birthday', 'truly', 'memorabl
e']
```

With this knowledge, we can create a custom function and apply it to a text message.

In [15]:
```
def text_cleaner(sentence):
    """Clean the text using typical NLP-Steps.

    Steps include: Lemmatization, removing stop words, removing punctuations

    Args:
        sentence (str): The uncleaned text.

    Returns:
        str: The cleaned text.

    """

    # Create the Doc object named `text` from `sentence` using `nlp()`
    doc = nlp(sentence)
    # Lemmatization
    lemma_token = [token.lemma_ for token in doc if token.pos_ != 'PRON']
    # Remove stop words and converting tokens to lowercase
    no_stopWords_lemma_token = [token.lower() for token in lemma_token if token not ir
    # Remove punctuations
    clean_doc = [token for token in no_stopWords_lemma_token if token not in punctuati
    # Use the `.join` method on `text` to convert string
    joined_clean_doc = " ".join(clean_doc)
    # Use `re.sub()` to substitute multiple spaces or dots`[\.\s]+` to single space `'
    final_doc = re.sub('[\.\s]+', ' ', joined_clean_doc)
    return final_doc

print(text_cleaner(df.loc[3202, "msg"]))
```

```
haha yup hopefully lose kg mon hip hop go orchard weigh
```

Now we have prepared the text data, we can vectorize it. In vectorization, text data is represented in numerical form. We can use the *Bag of Words* (BoW) and *Term Frequency-Inverse Document Frequency* (TF-IDF) methods for this with the help of `sklearn`.

Run the following cell to prepare the data with `text_cleaner()`. You can use the `my_Series.apply()` method to do this. It applies a function to each value of a `Series`. We then divide the data set into a training and test set. This may take a moment.

In [16]:
```
df.loc[:, 'msg_clean'] = df.loc[:, 'msg'].apply(text_cleaner)  # clean the data

# split data into train and test features and targets
from sklearn.model_selection import train_test_split
features = df.loc[:, 'msg_clean']
```

```
target = df.loc[:, 'status']
features_train, features_test, target_train, target_test = train_test_split(features,
                                                                            target,
                                                                            test_size=
                                                                            random_sta
```

BoW takes all the unique words present in the corpus and represents each text based on how often a word appears in that text. To apply *BoW* vectorization, we use `CountVectorizer` from `sklearn.feature_extraction.text`. This is a *transformer* (like `StandardScaler`). It can therefore be fitted (`my_transformer.fit()`) and applied (`my_tranformer.transform()`). In the following cell we convert both directly for the training data (`my_transformer.fit_transform()`).

In [17]:
```
from sklearn.feature_extraction.text import CountVectorizer
count_vectorizer = CountVectorizer()
features_train_bow = count_vectorizer.fit_transform(features_train)
```

With `my_transformer.get_feature_names()` we get the unique words in the corpus, which we can use as column names. Most messages use only a small part of these features. Then we end up with what's called a *sparse matrix*. This is a matrix that contains the value 0 a lot. To store them efficiently in the memory, you can use the datatype `.csr_matrix` (*compressed sparse row matrix*), which originally comes from the `scipy` module. `sklearn` can handle this like an array or a `DataFrame`. However, in order to be able to take a look at it, we'll convert it into a `DataFrame`.

In [18]:
```
bow_features = count_vectorizer.get_feature_names() # get names of new features

# transform vectorized data into DataFrame
bow_array = features_train_bow.toarray()
bow_vector = pd.DataFrame(bow_array, columns=bow_features)
bow_vector.head()
```

Out[18]:

	00	000	008704050406	0121	0125698789	02	0207	02072069400	02073162414	02085076972
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0

5 rows × 6333 columns

In the next step we'll see how often the `call` value occurs in the three messages 3, 7 and 11.

In [19]:
```
print(bow_vector.loc[[3,7,11],"call"])
print("Message 3: ", features_train.iloc[3])
print("Message 7: ", features_train.iloc[7])
print("Message 11: ", features_train.iloc[11])
```

```
3     1
7     1
11    3
Name: call, dtype: int64
Message 3:  double mins 1000 txt orange tariff latest motorola sonyericsson nokia blu
etooth free call mobileupd8 08000839402 call2optout hf8
Message 7:  work please call
Message 11:  santa call would little one like call santa xmas eve call 09077818151 bo
ok time calls1 50ppm last 3min 30 t&c www santacalling com
```

Unlike the BoW method, the *Term Frequency Inverse Document Frequency* (TF-IDF) vectorization takes into account the *relevance* of a word.

To determine the relevance of a word, TF-IDF takes two things into account, as its name suggests:

1. How often a **word** appears in an **individual message**, based on the number of words in the same **message** (*term frequency*, the **TF** in **TF-IDF**).

$$\mathrm{TF}\left(\textbf{word}, \textbf{message} \right) = \frac{\mathrm{number\, of\, instances\, of\, a\, word\, \textbf{word} \,in\, a\, \textbf{message}}}{\mathrm{number\, of\, all\, words\, in\, the\, same\,\textbf{message}}}$$

1. How often a **word** appears in the entire **corpus**, in relation to the **size of the corpus** (*inverse document frequency*, the **IDF** in **TF-IDF**).

$$\mathrm{IDF}\left(\textbf{word}, \textbf{corpus}\right) = \ln\left(\frac{\mathrm{number \, of \,\textbf{messages}\, in\, the \, \textbf{corpus}}}{\mathrm{number\, of \, \textbf{messages}, \, that\, contain\, the \, \textbf{word}}}\right)$$

This is a *natural logarithm*. The **TF-IDF** value of a **word** in a **message** in the **corpus** is defined as the product of **TF** and **IDF**.

$$\mathrm{TF\text{-}IDF} (\textbf{word},\textbf{message}, \textbf{corpus}) = \mathrm{TF}(\textbf{word},\textbf{message}) \,\cdot \, \mathrm{IDF}(\textbf{word}, \textbf{corpus})$$

The more **messages** the **word** appears in, the less valuable this **word** is for differentiating between text types. The **TF-IDF** of this kind of **word** would be small. An important **word** would be one that occurs very rarely in the entire **corpus**.

In *Chapter 4, Vectorizing Text* we calculated some TF-IDF values as examples, and `sklearn` offers `TfidfVectorizer` for this purpose. We can use it like `CountVectorizer`:

In [20]:
```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer()
features_train_tfidf = tfidf_vectorizer.fit_transform(features_train)
```

Since `features_train_tfidf` is a `csr_matrix`, we'll convert it to a `DataFrame` to understand it better.

In [21]:
```
tfidf_features = tfidf_vectorizer.get_feature_names()  # get names of new features

# transform vectorized data into DataFrame
tfidf_vector = pd.DataFrame(features_train_tfidf.toarray(), columns=tfidf_features)
tfidf_vector.head()
```

Out[21]:

	00	000	008704050406	0121	0125698789	02	0207	02072069400	02073162414	02085076972
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 6333 columns

Let's look at the *BoW* and TF-IDF values of the first messages 3 and 7 for the word `call`:

In [22]:
```
print(bow_vector.loc[[3,7], 'call'])
print(tfidf_vector.loc[[3,7], 'call'])
```

```
3    1
7    1
Name: call, dtype: int64
3    0.119834
7    0.437615
Name: call, dtype: float64
```

`'call'` only appears once in both these messages. However, the TF-IDF value is much higher for message 7. This is because there are fewer words in the message, so `'call'` becomes more important for this message.

After vectorization, we could use the text data as usual with a classifier to make predictions based on it.

Congratulations: Now you have refreshed your knowledge of lemmatization, removing stop words and removing punctuation marks. You have also recapped how to vectorize text data.

Remember:

- The SVM classifies data with decision lines and planes by using support vectors.
- `SVC` knows the kernels `'linear'`, `'poly'` and `'rbf'` by default.
- Some common text cleaning tasks for NLP are lemmatization, stop word removal, and punctuation removal.
- The BoW method (`CountVectorizer`) vectorizes words based on their frequencies in the text.
- TF-IDF vectorization (`TfidfVectorizer`) takes into account the *relevance* of a word.

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

Found a mistake? Contact Support at support@stackfuel.com.

The data in this chapter was used here first: Almeida, T.A., Gomez Hidalgo, J.M., Yamakami, A. Contributions to the Study of SMS Spam Filtering: New Collection and Results. Proceedings of the 2011 ACM Symposium on Document Engineering (DOCENG'11), Mountain View, CA, USA, 2011.