Feature Engineering with Polynomials

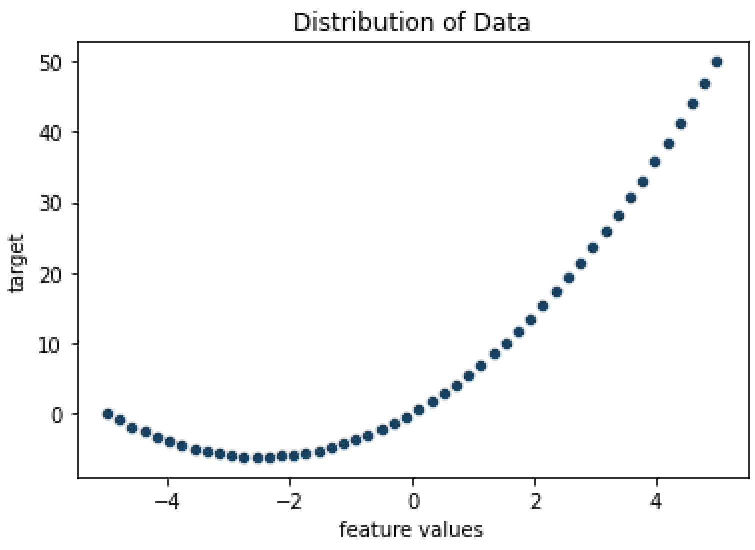
Module 1 | Chapter 4 | Notebook 2

A widespread problem with predictions based on statistical models is what's called underfitting, i.e. the opposite of overfitting, see *Module 1, Chapter 1*). Among other things, this means that the predictions do not become more accurate, even though you use more and more data to train the model. Very often the reason for this is that the model is too simple. For example, let's imagine that have a data set that has the following distribution (simply run the following cell)

In [1]:
```
import numpy as np
import seaborn as sns # import seaborn to make a simple scatterplot

features = np.linspace(start=-5, stop=5, num=50) # linspace creates num values evenly
target = features**2 + 5*features # a possible dependence of the target on the feature

ax = sns.scatterplot(x=features,
                     y=target,
                     color='#17415f')
ax.set(title='Distribution of Data', #style
       xlabel='feature values',
       ylabel='target');
```



Suppose we want to model this data with a linear regression. This would mean that we apply a straight line to the data set. Obviously this is not a good model and you would say that the linear regression model is underfitted to the data or that a straight line is far too simple a model for the data set. So what can you do to solve this problem? Here's an idea: We give the straight line the ability to bend! This idea is the foundation of feature engineering and polynomial regression - making a simple model more complex to take any nonlinearity in the data set into account. In this lesson we'll have a look at how this works. By the end of this lesson you will be able to:

- Understand the term feature engineering
- Create features of a higher degree
- Carry out a polynomial regression

An initial prediction

Scenario: 1001Wines is an online retailer that sells wines through its website. Recently, 1001Wines took over a start-up company that specializes in producing wines synthetically. They want to use this expertise to create a separate product range. The customers of 1001Wines leave ratings for the products. This data will be used to predict how well new wines will be accepted by customers in the future.

Since you already created a good product recommendation algorithm for 1001Wines (see *Module 0, Chapter 2*), they came back to you for more help. They want you to analyze the composition of the wines in view of their ratings. They provided you with the respective wine characteristics in the file *wine_composition.csv*. The relevant ratings are in the file *clustering_data-2.csv*. Both files are already in the current working directory.

We'll start by importing and looking at the data. First import the data in *wine_composition.csv* and store it as a `DataFrame` named `df_composition`. Then print the first five rows.

In [3]:
```
import pandas as pd
df_composition = pd.read_csv('wine_composition.csv')
df_composition
```

Out[3]:

	fixed.acidity	volatile.acidity	citric.acid	residual.sugar	chlorides	free.sulfur.dioxide	total.sulfur.
0	7.0	0.270	0.36	20.7	0.045	45.0	
1	6.3	0.300	0.34	1.6	0.049	14.0	
2	8.1	0.280	0.40	6.9	0.050	30.0	
3	7.2	0.230	0.32	8.5	0.058	47.0	
4	7.2	0.230	0.32	8.5	0.058	47.0	
...	...	...	...	...	...	...	
6492	6.2	0.600	0.08	2.0	0.090	32.0	
6493	5.9	0.550	0.10	2.2	0.062	39.0	
6494	6.3	0.510	0.13	2.3	0.076	29.0	
6495	5.9	0.645	0.12	2.0	0.075	32.0	
6496	6.0	0.310	0.47	3.6	0.067	18.0	

6497 rows × 13 columns

You can see that the data contains the different chemical properties of all the wines. The following data dictionary explains what the data means.

Column number	Column name	Type	Description
0	`'fixed.acidity'`	continuous (`float`)	liquid acid content.
1	`'volatile.acidity'`	continuous (`float`)	gaseous acid content, particularly relevant for the odor.
2	`'citiric.acid'`	continuous (`float`)	citric acid content. Part of the liquid acid content. Affects freshness in taste.
3	`'residual.sugar'`	continuous (`float')	Natural sugar of the grapes, which is still present at the end of fermentation stage.
4	`'chlorides'`	continuous (`float`)	chloride content. Minerals, which can be dependent on the wine growing region.
5	`'free.sulfur.dioxide'`	continuous (`float`)	free sulfur dioxide. Perceptible by smell.
6	`'total.sulfur.dioxide'`	continuous (`float`)	total sulfur dioxide content
7	`'density'`	continuous (`float`)	density of the wine
8	`'PH'`	continuous (`float`)	pH value. Determined by the acid content.
9	`'sulphates'`	continuous (`float`)	sulphate content.
10	`'alcohol'`	continuous (`float`)	alcohol content
11	`'color'`	categorical	color of the wine. Contains only `'red'` and `'white'`
12	`'id'`	categorical (`int`)	Unique identification number of the wine.

Which data types do the entries of the columns in `df_composition` have?

In [4]:
```
df_composition.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6497 entries, 0 to 6496
Data columns (total 13 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   fixed.acidity         6497 non-null   float64
 1   volatile.acidity      6497 non-null   float64
 2   citric.acid           6497 non-null   float64
 3   residual.sugar        6497 non-null   float64
 4   chlorides             6497 non-null   float64
 5   free.sulfur.dioxide   6497 non-null   float64
 6   total.sulfur.dioxide  6497 non-null   float64
 7   density               6497 non-null   float64
 8   pH                    6497 non-null   float64
 9   sulphates             6497 non-null   float64
 10  alcohol               6497 non-null   float64
 11  color                 6497 non-null   object 
 12  id                    6497 non-null   int64  
dtypes: float64(11), int64(1), object(1)
memory usage: 660.0+ KB
```

`'color'` is interpreted as a `str` If you want, you can convert this column into a categorical column in the following cell.

In [5]:
```
df_composition.loc[:, 'color'] = df_composition.loc[:, 'color'].astype('category')
```

Are there missing values in the data?

In [7]:
```
df_composition.isna().sum()
```

Out[7]:
```
fixed.acidity           0
volatile.acidity        0
citric.acid             0
residual.sugar          0
chlorides               0
free.sulfur.dioxide     0
total.sulfur.dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
color                   0
id                      0
dtype: int64
```

There are no missing values. It looks like 1001Wines gave you structured and clean data!

Now import the *wine_rating.csv* file. Store the data as a `DataFrame` named `df_rating` and print the first 5 rows.

In [9]:
```
df_rating  = pd.read_csv('wine_rating.csv')
df_rating.head()
```

Out[9]:

	id	rating
0	0	6
1	1	6
2	2	6
3	3	6
4	4	6

`df_rating` contains the identification numbers of the products and their associated rating. Check whether missing values are included in the data.

In [15]:
```
df_rating.isna().sum()
```

Out[15]:
```
id        0
rating    0
dtype: int64
```

There are also no missing values here. Which possible values does the `'rating'` column contain or which unique numbers does it contain?

In [17]:
```
sorted(df_rating.loc[:,'rating'].unique())
```

Out[17]:
```
[3, 4, 5, 6, 7, 8, 9]
```

`'rating'` contains the values 3, 4, 5, 6, 7, 8, and 9. Your clients told you that the rating scale has 10 levels, with 10 being the best rating and 1 being the worst. The values in `'df_rating'` are integer values, as these are not average values but median values of all ratings of the respective product.

With this rating scale, the entries are actually ordinal categories. So they have an order: 2 is better than 1 and worse than 3. In practice, it's common to predict these values with a regression, which is actually used for continuous values. This is also the approach we'll take. However, it's important to be aware that customers did not necessarily perceive the ratings as numbers when they chose them. On a 10 star scale, 5 feels like it's in the middle and 1 feels very bad. This does not necessarily mean that the 5 is 5 times as good as the 1. So the values are ordinal.

First, we'll use a normal linear regression to get a feel for how good our predictions already are. Import `LinearRegression` from `sklearn.linear_model` and instantiate the model with the name `model`.

In [18]:
```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
```

For that we need the feature matrix and the target vector. To do this, we first need to know whether the `'id'` column in both DataFrames matches up. If this is not the case, we would have to merge the data so that the rows refer to the same wines. How many values don't match? To answer this, you can compare the columns, therefore creating an array containing the values `True` and `False`. If you find the sum of this array, `True` will be counted as a `1` and `False` will be interpreted as `0`. Try it out.

In [22]:
```
df_rating.loc[:,'id'] != df_composition.loc[:,'id']

sum(df_composition.loc[:, 'id'] != df_rating.loc[:, 'id'])
```

Out[22]:
```
0
```

We're in luck. All of the identification numbers match. So we can use the `'rating'` column directly as a target vector. Call this `target`. Use all the columns from `df_composition` except for `'color'` and `'id'` as features and store them as `features`. This data will form the basis for our *baseline model*.

In [23]:
```
target = df_rating.loc[:, 'rating']
features = df_composition.drop(['color','id'], axis=1)
```

This allows us to get started with an initial prediction. Now we just need to think about which evaluation metric we want to use. As we are performing a regression, we will predict continuous values that never exactly match the actual ratings. So it would be interesting to know how far off we are on average. We can obtain this information with the mean **absolute** error.

To get a realistic impression of this very simple model, it is best to use `cross_val_score` from `sklearn.model_selection` (link to documentation). You have already used this function in *Introduction to Pipelines (Module 1 Chapter 2)*. It uses the following parameters:

```
cross_val_score(estimator=model,         # the model that will make the 
predictions
                X=features            # the feature matrix
                y=target               # the target vector
                cv=int                   # the number of folds to be used 
for cross-validation
                scoring=str or function # the metric that rates the result
               )
```

Use the values `cv=5` and `scoring='neg_mean_absolute_error'`. You get back the negative mean absolute errors for each of the 5 validations. A higher value therefore stands for a better model. All *scoring* functions of `cross_val_score` are made this way. Calculate the mean value and print it.

In [25]:
```
from sklearn.model_selection import cross_val_score
cv_results = cross_val_score(estimator=model, X=features, y=target, cv=5, scoring='neg
cv_results.mean()
```

Out[25]:
```
-0.58104275094183
```

We are generally just over half a rating point off (at about -0.58). That really isn't so bad.

Congratulations: You have got to know the data and carried out a quick regression based on it. The results are already a solid foundation. You can often improve on the results by adding more features to it. We'll now use feature engineering to do this.

Creating features of a higher degree

Creating new features from those you already have is called feature engineering. You have already carried out feature engineering at various points in the course, e.g. in

- Automatically Detecting Room Occupancy (Module 1 Chapter 2)*, we created the feature `'msm'`.

Good features are the basis of a good prediction. They can greatly improve a model's ability to make predictions. There are no limits to your imagination when creating new features. As a result, it can also take a lot of time. Therefore it can help a lot if you are able to use your own domain knowledge. It is often a good idea to get help from people in the relevant specialist department. Together, you might be able to find out more quickly which new features are more significant. But it's also possible that there isn't much more you can tease out of the data. If you feel stuck when feature engineering, this may be one of those cases. Don't spend too much time on it, even if it's tempting.

If you have no idea which features can help, it's often a good idea to look at the interactions and higher degrees of features. This means the paired products of the features and for example the squared values of the features in pairs. The linear regression generates the predictions just by weighting and summing up the features. It is therefore not able to create products from the features. For example, let's assume that any two features exist, we'll name them `feature_1` and `feature_2` here. Possible features of a higher degree would be $\left(feature\_1\right) \cdot \left(feature\_2\right)$ or $\left(feature\_1\right)^2$.

Creating higher-degree features by hand can be very tedious if you have a lot of features. Fortunately `sklearn` offers us the appropriate *transformer* for this. Feature engineering is part of data preparation. That's why the *transformer* is located in `sklearn.preprocessing`. These are `PolynomialFeatures` with the following parameters (link to documentation):

```
PolynomialFeatures(degree=int,           # The degree of the resulting polynomial.
                   interaction_only=bool # Controls whether self interactions are included.
                   include_bias=bool     # Controls whether the 1 is also included as a feature.
                  )
```

The most important hyperparameter of `PolynomialFeatures` is `degree`. This specifies the degree the new features should have. `include_bias` specifies whether a column consisting only of ones should be included. Normally we don't need this, because `LinearRegression` already takes care of this internally (with the `fit_intercept` parameter). The parameter `interaction_only` controls whether only the interactions should be generated. For example, $\left(feature\_1\right) \cdot \left(feature\_2\right)$ would be an interaction, while $\left(feature\_1\right)^2$ would not.

The following table should give you a feeling which features you get depending on the parameters if you have only 2 features $a$ and $b$:

`degree`	`interaction_only=False`	`interaction_only=True`
0	1	1
1	1, $a$, $b$	1, $a$, $b$
2	1, $a$, $b$, $a^2$, $a\cdot b$, $b^2$	1, $a$, $b$, $a\cdot b$
3	1, $a$, $b$, $a^2$, $a\cdot b$, $b^2$, $a^3$,$a^2\cdot b$, $a\cdot b^2$, $b^3$	1, $a$, $b$, $a\cdot b$

The 1 only occurs if `include_bias=True`. As you can see, the number of features increases very quickly if you don't only create the interactions. If you have more than two features, the number will increase even more quickly.

With real data, the predictive power of the regression model doesn't usually increase by such an extreme amount. Nevertheless, you can improve predictions a little in this way relatively easily, which can be a dramatic improvement depending on the context (for example, in competitions, it's not uncommon for the seventh decimal place to make all the difference). Let's take a look at how much that will help in the wine ratings.

To do this, import `PolynomialFeatures` from `sklearn.preprocessing`. Instantiate the *transformer* with the parameters `degree=2` and `include_bias=false` with the name `poly_transformer`.

Important: If you generate a lot of features with a higher degree, you very quickly run the risk of your polynomial regression model suffering from overfitting. Overfitting is covered in the video *The Overfitting Problem (Module 1 Chapter 1)*.

In [26]:
```
from sklearn.preprocessing import PolynomialFeatures
poly_transformer = PolynomialFeatures(degree=2, include_bias=False)
```

It's best to use `poly_transformer` as part of a `Pipeline` together with `model`. You learned about *Pipelines* in *Introduction to Pipelines (Module 1 Chapter 2)*. Import `Pipeline` from `sklearn.pipeline` Create a `Pipeline` called `pipeline`, made up of `poly_transformer` and `model`. Name the two steps within `pipeline` `'poly'` and `'reg'`.

In [27]:
```
from sklearn.pipeline import Pipeline
pipeline = Pipeline([('poly', poly_transformer),
                     ('reg', model)]
                   )
```

Now use `pipeline` to create and evaluate new predictions. Use `cross_val_score` again with the parameters `estimator=pipeline, X=features, y=target, cv=5, scoring='neg_mean_absolute_error'`. Calculate the mean of the model quality metric values and print it.

```
In [28]: cv_results = cross_val_score(estimator=pipeline,
                                      X=features,
                                      y=target,
                                      cv=5,
                                      scoring='neg_mean_absolute_error'
                                     )
         cv_results.mean()
```

```
Out[28]: -0.5771691803265556
```

Our predictions have improved by about `0.004` on average. This may not sound like too much, but note that the amount of work involved was very small, so it is often preferable to use polynomial regression rather than linear regression.

Congratulations: You have carried out a polynomial regression with the help of feature engineering. By creating new features, you were able to improve the predictions slightly, although we did not apply any special domain knowledge in this field. However, creating new features can also lead to overfitting. In the next lesson, you will learn about dimensionality reduction. You can use this to reduce the number of features without losing much information.

Remember:

- Feature engineering describes the process of creating new features that can help when making predictions
- Create features of a higher degree with the `PolynomialFeatures` transformer.
- A linear regression with features of a higher degree is a polynomial regression.

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 was published here first: P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 47(4):547-553, 2009.
