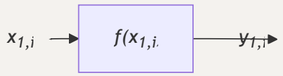
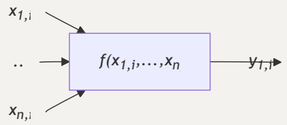
LINEARE REGRESSION

Joern Ploennigs · AI4SC

Regressionsmodelle sind eine Klasse von statistischen Modellen zum überwachten Lernen, die verwendet werden, um die Beziehung zwischen einer abhängigen Variable und einer oder mehreren unabhängigen Variablen zu modellieren. Sie werden häufig eingesetzt, um Vorhersagen zu treffen, Muster in Daten zu erkennen oder als Basismodell zum Vergleich. Zum Beispiel zur Vorhersage der Betonfestigkeit in Abhängigkeit von Mischungsverhältnis und Aushärtungszeit, zur Schätzung der Setzung eines Fundaments basierend auf Bodenkennwerten oder zur Prognose von Energieverbrauch eines Gebäudes.

Wir wollen verschiedene univariate und multivariate Modelle behandeln. Hierbei unterscheidet sich die Anzahl der Eingangsvariablen. Modelle mit einer Eingangsvariable werden *univariat* genannt und Modelle mit mehreren Eingangsvariablen heißen *multivariat*. Dies ist wichtig, da in der Praxis viele Ergebnisse nicht nur von einem Faktor ab hängen. Zum Beispiel, die Aushärtung von Beton hängt von Temperatur, Luftfeuchtigkeit und Mischung ab – ein multivariates Modell kann das berücksichtigen.





UNIVARIATE LINEARE REGRESSION

Eines der einfachsten Machine Learning Modelle ist die *Lineare Regression*. Es basiert auf der Idee einer linearen Funktion aus der linearen Algebra. Dort ist eine lineare Funktion definiert als:

$$\hat{y}_i = \hat{f}(x_i) = \beta_0 + \beta_1 \cdot x_i.$$

Die Funktion modelliert Beziehung zwischen der Zielvariable $\hat{y}_i$ und einer unabhängigen Variable x_i in Form einer einfachen Linie definiert durch die Funktion $\hat{f}(x_i)$. Die Parameter sind die y-Achsenabschnitt (auch: Schnittpunkt mit der y-Achse) (en. intercept) β_0 und der Anstieg β_1.

Wenn man z.B. weiß, dass bei einem Beton mit einer 0.45 Wasser-Zement-Wert eine Druckfestigkeit von 45 MPa erreicht wird, und bei einem 0.55 Wasser-Zement-Wert nur 35 MPa, kann ein lineares Modell helfen, diese Beziehung zu approximieren.

Bestimmen der Parameter für zwei Punkte

Aus der linearen Algebra wissen wir, dass wir die Parameter β_0 und β_1 bestimmen können (*en. fitting*), wenn wir zwei Punkte $[y_0, x_0]$ und $[y_1, x_1]$ kennen und das Gleichungssystem für diese aufstellen und lösen

$$y_0 = \beta_0 + \beta_1 \cdot x_0,$$
$$y_1 = \beta_0 + \beta_1 \cdot x_1.$$

Wir stellen y_0 nach β_0 um

$$\beta_0 = y_0 - \beta_1 \cdot x_0$$

und setzten es in y_1 ein und erhalten

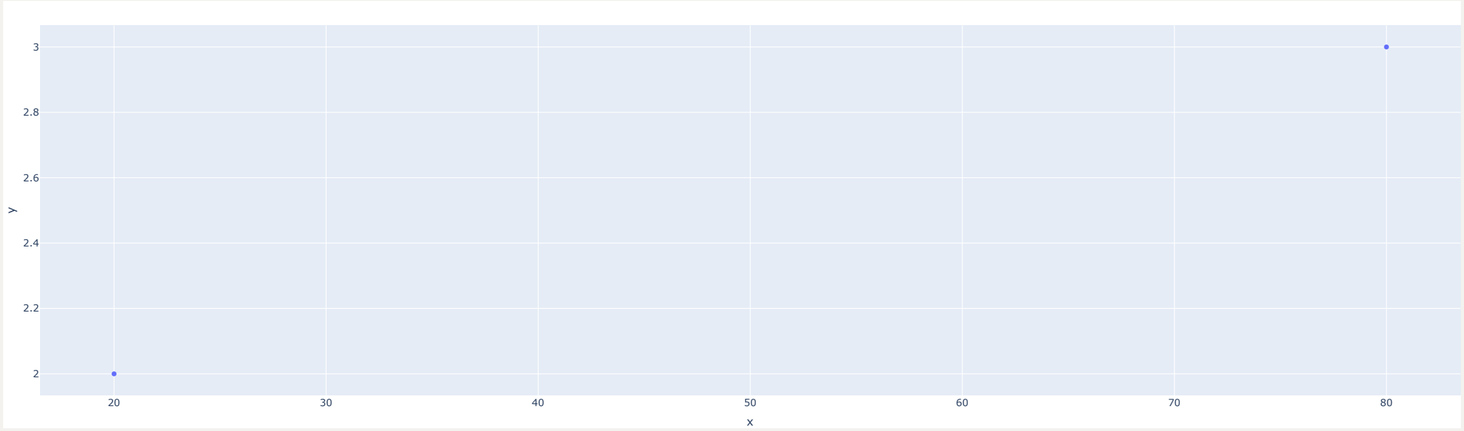
$$
\begin{aligned}
y_1 &= y_0 - \beta_1 \cdot x_0 + \beta_1 \cdot x_1, \\
y_1 - y_0 &= \beta_1 (x_1 - x_0), \\
\beta_1 &= \frac{y_1 - y_0}{x_1 - x_0}.
\end{aligned}
$$

Wenn wir das in β_0 einsetzen, folgt:

$$
\begin{aligned}
\beta_0 &= y_0 - \beta_1 \cdot x_0. \\
\beta_0 &= y_0 - \frac{y_1 \cdot x_0 - y_0 \cdot x_0}{x_1 - x_0}, \\
\beta_0 &= \frac{y_0 \cdot x_1 - y_0 \cdot x_0}{x_1 - x_0} - \frac{y_1 \cdot x_0 - y_0 \cdot x_0}{x_1 - x_0}, \\
\beta_0 &= \frac{y_0 \cdot x_1 - y_1 \cdot x_0}{x_1 - x_0}.
\end{aligned}
$$

Damit können wir die Parameter für einen Datensatz mit zwei Punkten bestimmen

```
import numpy as np # Import von NumPy
import pandas as pd # Import von Pandas
import plotly.express as px # Import von Plotly

xy = pd.DataFrame({"x":[20,80], "y":[2,3]})
px.scatter(xy, x="x", y="y")
```

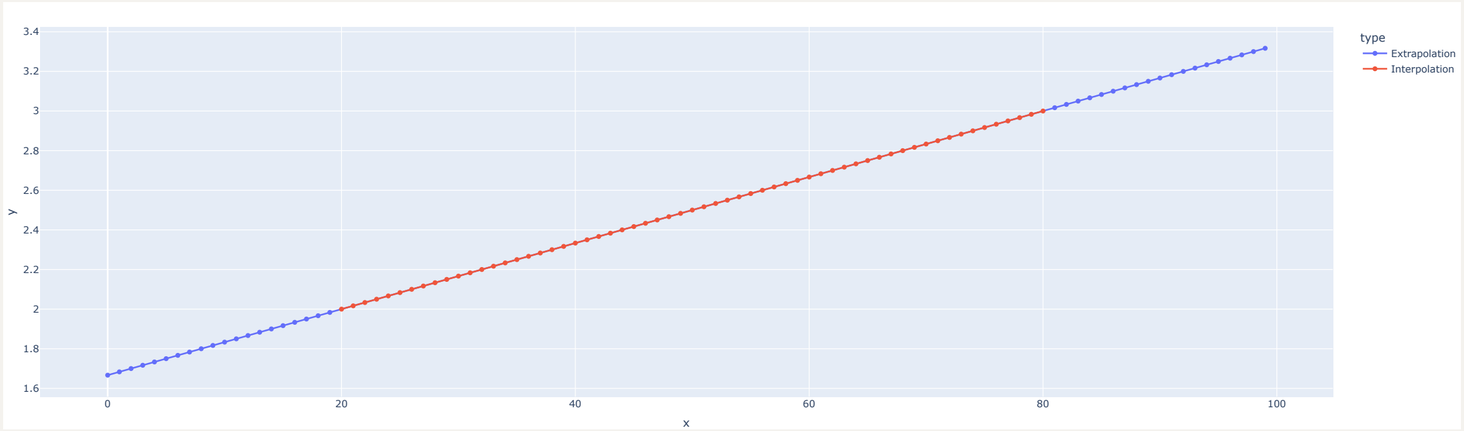



```
beta_1 = (xy.y[1] - xy.y[0]) / (xy.x[1] - xy.x[0])
beta_0 = xy.y[0] - beta_1 * xy.x[0]
beta_0, beta_1
```

```
(np.float64(1.6666666666666667), np.float64(0.016666666666666666))
```

Jetzt haben wir ein einfaches Modell bestimmt und wir können damit beliebige Punkte auf dieser Linie berechnen. Sind wir innerhalb des Bereiches $20 \leq x \leq 80$ so nennt man das *Interpolation*. Sind wir außerhalb $x < 20$ oder $x > 80$ so nennt man es *Extrapolation*.

```
xy = pd.DataFrame({"x":range(100)})
xy["y"]= beta_0 + beta_1 * xy.x
xy["type"] = np.where((20 <= xy.x) & (xy.x <= 80), "Interpolation", "Extrapolation")

px.line(xy, x="x", y="y", color="type", markers=True)
```



Bestimmen der Parameter für mehrere Punkte

Um die Parameter β_0 und β_1 zu schätzen, verwenden wir die Methode der kleinsten Quadrate (en. Ordinary Least Squares - OLS). Sie wurde von Carl Friedrich Gauß entwickelt zum Annähern von Planetenbahnen. OLS minimiert die Summe der quadrierten Anpassungsfehler. Der *Anpassungsfehler* ε_i berechnet sich aus der Differenz des Zielwertes y_i und der Vorhersage des linearen Modells y_i

$$\varepsilon_i = \hat{y}_i - y_i = \beta_0 + \beta_1 x_i - y_i.$$

Dieser Anpassungsfehler wird auch als *Residuum* bezeichnet. Man schreibt deshalb das lineare Modell auch oft als

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i.$$

Zu beachten ist, dass y_i der Zielwert ist und nicht die Vorhersage $\hat{y}_i$ von zuvor.

Hierbei wird angenommen, das der Fehler ε_i *normalverteilt* ist. Unter dieser Annahme lassen sich die Koeffizienten β_0 und β_1 mit der kleinsten Summe der Fehlerquadrate aus folgendem Optimierungsproblem bestimmen

$$\min_{\beta_0,\beta_1} \sum_{i=1}^{n} \varepsilon_i^2.$$

Aus der Mathematik ist bekannt, das eine Funktion ein lokales Extremum erreicht, wenn ihre erste Ableitung zu Null wird. Wir können also das Minimum bestimmen, indem wir die erste Ableitung gleich Null setzen. Da β_0 und β_1 unsere variablen Parameter sind, bilden wir hierfür die partielle Ableitung nach β_0 und β_1 und bestimmen davon die Nullstellen. Dabei hilft uns das Fehlerquadrat, dass es dadurch eine eindeutige Lösung gibt, welchje gegeben ist durch das lineare Gleichungssystem

$$n \cdot \beta_0 + \beta_1 \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$$

$$\beta_0 \sum_{i=1}^{n} x_i + \beta_1 \sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} x_i y_i$$

mit der Lösung für den Anstieg

$$\beta_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

mit der Summe der Abweichungsprodukte im Zähler und der Summe der Abweichungsquadrate im Nenner. Für die Abweichung ergibt sich

$$\beta_0 = \bar{y} - \beta_1 \bar{x}$$

mit den Mittelwerten für $\bar{x}$ und $\bar{y}$.

Wenn wir die Lösung vergleichen mit der Lösung für eine lineare Funktion mit zwei Punkten, so sehen wir eine Ähnlichkeit, insbesondere bei β_0, nur dass wir hier mit Mittelwerten $\bar{x}$ und $\bar{y}$ statt direkt mit dem Vorgängerwerten rechnen.

Beispiel

Wir wollen das lineare Modell nutzen, um den Zusammenhang zwischen Außentemperatur und dem Energieverbrauch zu modellieren. Hierfür laden wir wieder den Energie- und Wetterdatensatz der Uni Rostock

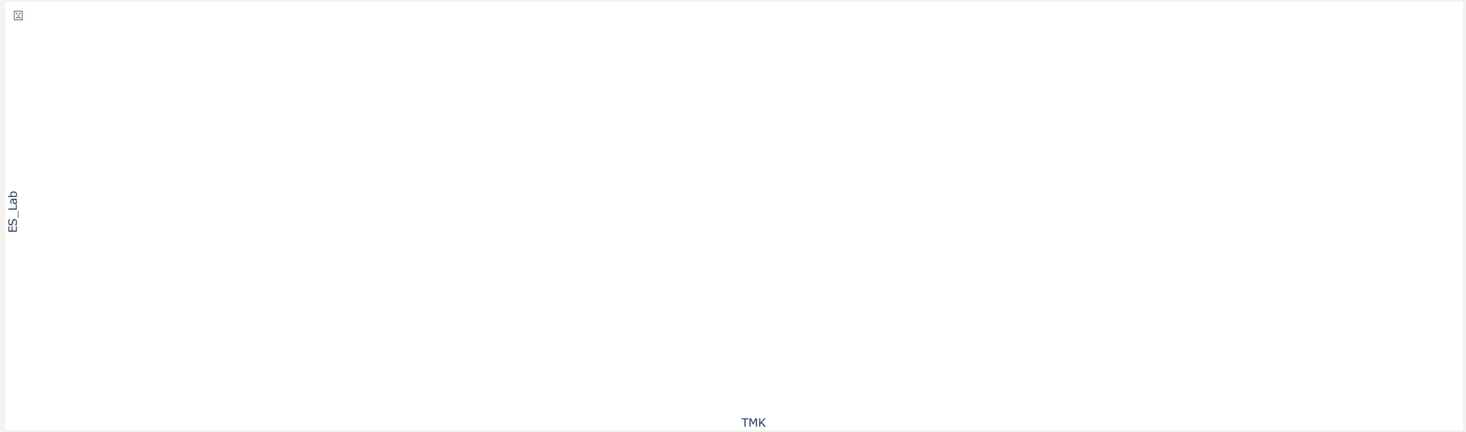
```
egywth = pd.read_csv("../data/UROS/Energy1D_weather_clean.csv", parse_dates=[0])
egywth.head()
```

	Date	EV_HT_740	EV_NT_740	E_AV_Lab	E_SV_Lab	ES_Lab	DATUM_DT	STATIONS_ID	MESS_DATUM	QN_3	...	TMK	UPM	TXK	TNK	TGK	eor	TemperaturKlasse	HeizKuehlTage
0	2020-12-30 00:00:00+00:00	NaN	NaN	NaN	NaN	NaN	2020-12-30 00:00:00+00:00	4271.0	20201230.0	10.0	...	4.0	81.0	4.6	2.9	2.2	eor	Cold	Heizgradtag
1	2020-12-31 00:00:00+00:00	NaN	NaN	1256.0	291.0	5.0	2020-12-31 00:00:00+00:00	4271.0	20201231.0	10.0	...	3.4	83.0	4.4	2.3	0.9	eor	Cold	Heizgradtag
2	2021-01-01 00:00:00+00:00	0.0	4080.0	1221.0	290.0	1.0	2021-01-01 00:00:00+00:00	4271.0	20210101.0	10.0	...	2.0	90.0	3.0	1.1	0.3	eor	Cold	Heizgradtag
3	2021-01-02 00:00:00+00:00	1170.0	2630.0	1243.0	284.0	2.0	2021-01-02 00:00:00+00:00	4271.0	20210102.0	10.0	...	3.3	95.0	4.0	2.6	2.0	eor	Cold	Heizgradtag
4	2021-01-03 00:00:00+00:00	0.0	3750.0	1222.0	283.0	2.0	2021-01-03 00:00:00+00:00	4271.0	20210103.0	10.0	...	3.5	81.0	4.6	2.4	0.8	eor	Cold	Heizgradtag

5 rows × 30 columns

Wir interessieren uns als erstes den Energieverbrauch `ES_Lab` und die mittlere Außentemperatur `TMK` . Wir betrachten uns das Streudiagramm. Darin lässt sich ein leichter linearer Trend erkennen, da der Energieverbrauch bei hohen Temperaturen über 10 Grad steigt, was auf eine aktive Kühlung hinweist.

```
fig = px.scatter(egywth, x="TMK", y="ES_Lab", opacity=.5)
fig
```



Mit den Formeln oben können wir uns jetzt ein lineares Modell berechnen. Hierbei ist `ES_Lab` unsere Zielvariable $\boldsymbol y$ und `TMK` unsere Eingangsvariable $\boldsymbol x$.

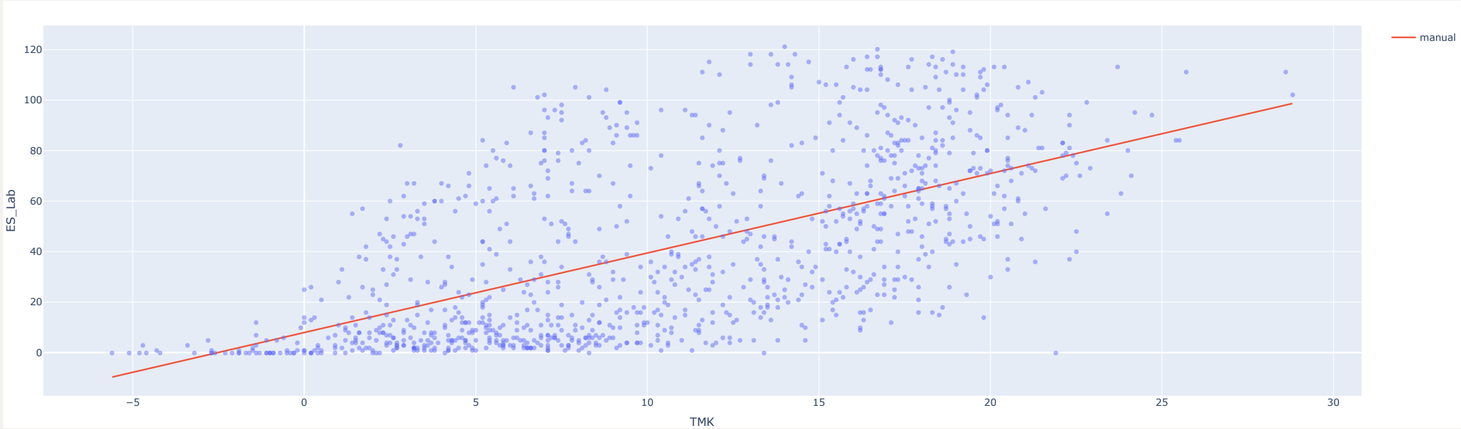
```
beta_1 = ((egywth.TMK - egywth.TMK.mean())*(egywth.ES_Lab - egywth.ES_Lab.mean())).sum()
beta_1 /= (egywth.TMK - egywth.TMK.mean()).pow(2).sum()

beta_0 = egywth.ES_Lab.mean() - beta_1 * egywth.TMK.mean()

beta_0, beta_1
```

```
(np.float64(7.95973736889691), np.float64(3.1496156006907596))
```

Um das Lineare Modell mit dem Originalwerten zu vergleichen, können wir die Vorhersagen entsprechend der ersten Formel bestimmen

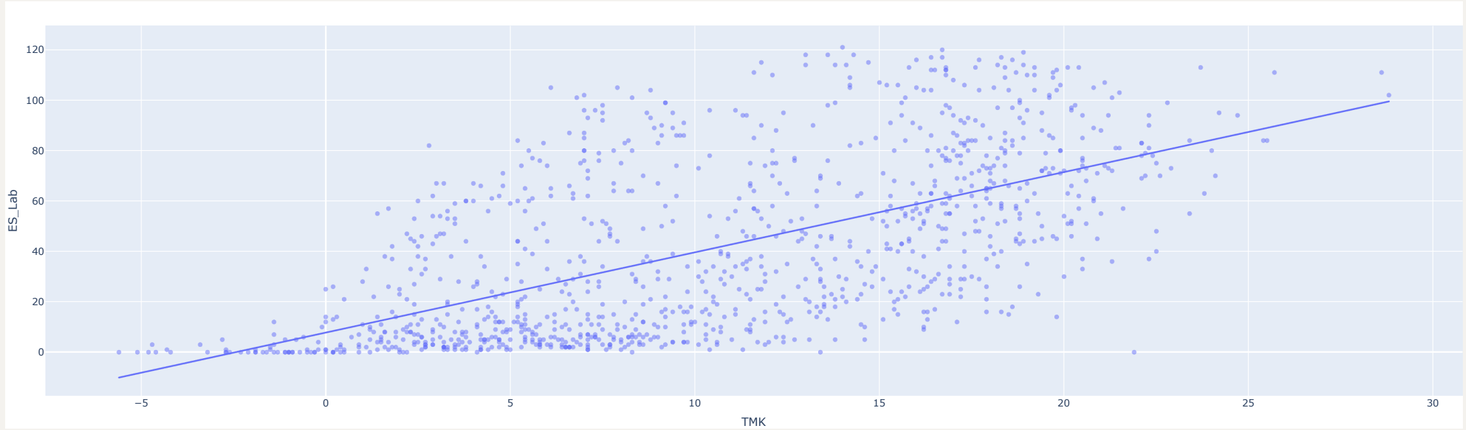
```
# Wir nehmen noch einmal die TMK Werte und sortieren sie
pred_x = np.sort(egywth.TMK.values)

pred_y = beta_0 + beta_1 * pred_x
```

```
fig.add_scatter(x=pred_x, y=pred_y, name="manual")
fig
```



Das können wir uns sogar im Streudiagramm mit Plotly direkt berechnen lassen, indem wir die Trendlinie `ols` angeben. Dieses Lineare Regressionsmodell wird von Plotly mit der Bibliothek `statsmodels` berechnet.

```
px.scatter(egywth, x="TMK", y="ES_Lab", opacity=.5, trendline="ols")
```



ML-Frameworks für Lineare Regression

SciKit-Learn

Die erste Bibliothek ist SciKit-Learn. Sie enthält viele ML-Modelle inklusive der Linearen Regression.

```
from sklearn.linear_model import LinearRegression

sklm = LinearRegression() # Das legt nur eine Instanz der Modellklasse an
```

Dieser Aufruf erstellt nur eine Modellinstanz aber enthält noch keine Daten. Diese müssen wir trainieren, was wir mit der Methode `fit()` machen können, der wir die Eingabedaten und die Zieldaten übergeben. SciKit-Learn arbeitet allerdings nicht direkt auf Pandas DataFrames sondern mit Numpy Arrays, weshalb wir die Pandas Serien in ein Numpy Arrays umwandeln müssen, wofür wir das Attribute `values` und die Methode `reshape` nutzen (um das Array in eine Matrix umzuwandeln).

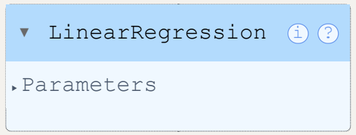
```
X = egywth.TMK.values.reshape(-1, 1) # Wir erstellen eine Matrix der Eingangsdaten
Y = egywth.ES_Lab.values             # Wir erstellen einen Vektor der Zieldaten
sklm.fit(X, Y)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[11], line 3
      1 X = egywth.TMK.values.reshape(-1, 1) # Wir erstellen eine Matrix der Eingangsdaten
      2 Y = egywth.ES_Lab.values             # Wir erstellen einen Vektor der Zieldaten
----> 3 sklm.fit(X, Y)

File ~/Documents/code/Lehre/IntroductionMachineLearning/.venv/lib/python3.12/site-packages/sklearn/base.py:1336, in _fit_context.<locals>.decorator.<locals>.wrapper(estimator, *args, **kwargs)
   1329     estimator._validate_params()
   1331 with config_context(
   1332     skip_parameter_validation=(
   1333         prefer_skip_nested_validation or global_skip_validation
   1334     )
   1335 ):
-> 1336     return fit_method(estimator, *args, **kwargs)

File ~/Documents/code/Lehre/IntroductionMachineLearning/.venv/lib/python3.12/site-packages/sklearn/linear_model/_base.py:630, in LinearRegression.fit(self, X, y, sample_weight)
    626 n_jobs_ = self.n_jobs
    628 accept_sparse = False if self.positive else ["csr", "csc", "coo"]
--> 630 X, y = validate_data(
    631     self,
    632     X,
    633     y,
    634     accept_sparse=accept_sparse,
    635     y_numeric=True,
    636     multi_output=True,
    637     force_writeable=True,
    638 )
    640 has_sw = sample_weight is not None
    641 if has_sw:

File ~/Documents/code/Lehre/IntroductionMachineLearning/.venv/lib/python3.12/site-packages/sklearn/utils/validation.py:2919, in validate_data(_estimator, X, y, reset, validate_separately, skip_check_array, **check_par
   2917         y = check_array(y, input_name="y", **check_y_params)
   2918     else:
-> 2919         X, y = check_X_y(X, y, **check_params)
   2920     out = X, y
   2922 if not no_val_X and check_params.get("ensure_2d", True):

File ~/Documents/code/Lehre/IntroductionMachineLearning/.venv/lib/python3.12/site-packages/sklearn/utils/validation.py:1331, in check_X_y(X, y, accept_sparse, accept_large_sparse, dtype, order, copy, force_writeable,
   1310     raise ValueError(
   1311         f"{estimator_name} requires y to be passed, but the target y is None"
   1312     )
   1314 X = check_array(
   1315     X,
   1316     accept_sparse=accept_sparse,
   (...)   1328     input_name="X",
   1329 )
-> 1331 y = _check_y(y, multi_output=multi_output, y_numeric=y_numeric, estimator=estimator)
   1333 check_consistent_length(X, y)
   1335 return X, y

File ~/Documents/code/Lehre/IntroductionMachineLearning/.venv/lib/python3.12/site-packages/sklearn/utils/validation.py:1341, in _check_y(y, multi_output, y_numeric, estimator)
   1339 """Isolated part of check_X_y dedicated to y validation"""
   1340 if multi_output:
-> 1341     y = check_array(
   1342         y,
   1343         accept_sparse=False,
   1344         ensure_all_finite=True,
   1345         ensure_2d=False,
   1346         dtype=None,
   1347         input_name="y",
   1348         estimator=estimator,
```

```
   1349     )
   1350 else:
   1351     estimator_name = _check_estimator_name(estimator)

File ~/Documents/code/Lehre/IntroductionMachineLearning/.venv/lib/python3.12/site-packages/sklearn/utils/validation.py:1074, in check_array(array, accept_sparse, accept_large_sparse, dtype, order, copy, force_writeabl
   1068     raise ValueError(
   1069         f"Found array with dim {array.ndim},"
   1070         f" while dim <= 2 is required{context}."
   1071     )
   1073 if ensure_all_finite:
-> 1074     _assert_all_finite(
   1075         array,
   1076         input_name=input_name,
   1077         estimator_name=estimator_name,
   1078         allow_nan=ensure_all_finite ==              ,
   1079     )
   1081 if copy:
   1082     if _is_numpy_namespace(xp):
   1083         # only make a copy if `array` and `array_orig` may share memory`

File ~/Documents/code/Lehre/IntroductionMachineLearning/.venv/lib/python3.12/site-packages/sklearn/utils/validation.py:133, in _assert_all_finite(X, allow_nan, msg_dtype, estimator_name, input_name)
    130 if first_pass_isfinite:
    131     return
--> 133 _assert_all_finite_element_wise(
    134     X,
    135     xp=xp,
    136     allow_nan=allow_nan,
    137     msg_dtype=msg_dtype,
    138     estimator_name=estimator_name,
    139     input_name=input_name,
    140 )

File ~/Documents/code/Lehre/IntroductionMachineLearning/.venv/lib/python3.12/site-packages/sklearn/utils/validation.py:182, in _assert_all_finite_element_wise(X, xp, allow_nan, msg_dtype, estimator_name, input_name)
    165 if estimator_name and input_name == "X" and has_nan_error:
    166     # Improve the error message on how to handle missing values in
    167     # scikit-learn.
    168     msg_err += (
    169         f"\n{estimator_name} does not accept missing values"
    170         " encoded as NaN natively. For supervised learning, you might want"
  (...)    180         "#estimators-that-handle-nan-values"
    181     )
--> 182 raise ValueError(msg_err)

ValueError: Input y contains NaN.
```

Hier gibt es allerdings einen Fehler. SciKit beschwert sich, dass der Datensatz `NaN` enthält. Zum Glück haben wir beim Data Cleaning gelernt, wie wir diese mit `.dropna()` entfernen können. Das müssen wir allerdings für den Eingangsvektor `X` und `Y` gleichzeitig machen, damit sie sich nicht verschieben oder unterschiedlich lang sind. Deshalb wenden wir `.dropna()` auf die Orginaldaten an und splitten die Daten danach in `X` und `Y`.

```
egywthNoNA=egywth[["TMK", "ES_Lab"]].dropna().sort_values("TMK") # Drop NA rows

# Modell trainieren
X = egywthNoNA.TMK.values.reshape(-1, 1)
Y = egywthNoNA.ES_Lab.values
sklm.fit(X, Y)
```



Wir können die gelernten Parameter uns anschauen mit

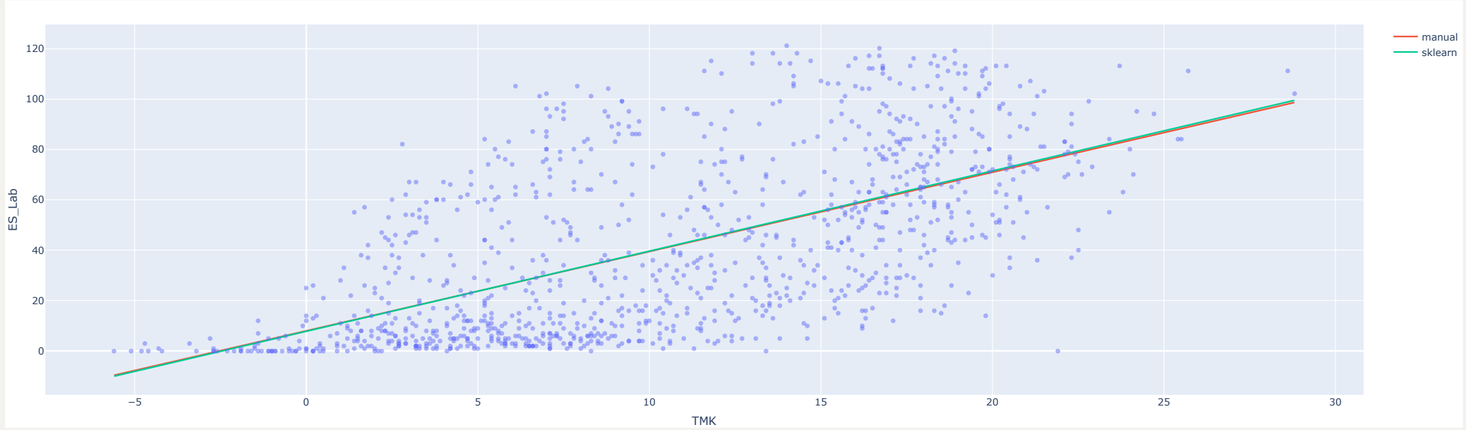
```
sklm.intercept_, sklm.coef_
```

```
(np.float64(7.778394959065665), array([3.18532247]))
```

Eine Vorhersage können wir bestimmen mit der Methode `predict` welcher wir die vorherzusagenden X-Matrix übergeben.

```
egywthNoNA["pred"] = sklm.predict(X)
```

```
fig.add_scatter(x=egywthNoNA.TMK, y=egywthNoNA.pred, name="sklearn")
fig
```



Wir erkennen eine Abweichung zwischen der manuell berechneten Funktion und der Funktion von SciKit. Das lässt sich mit den entfernten Werten erklären.

Statsmodels

Alternativ zu SciKit-Learn kann man Statsmodels nutzen. Es fokussiert stärker auf statistisch Modelle, einschließlich `ols`, *ordinary least squares*, d.h. lineare Regression. Die grundlegende Syntax von `ols` ist

```
smf.ols(equation, data)
```

Die Besonderheit ist, dass der Zusammenhang hier mit `equation` als Formel in der Form `"y ~ x"` angebbar ist und `data` direkt ein Dataframe ist, der die Variablen enthält. Es ist also nicht erforderlich, eine explizite Designmatrix zu erstellen, wie bei sklearn. Diese Formeln stammen aus ursprünglich aus der Programmiersprache R für maschinelles Lernen, wo sie ähnlich verwendet werden.

In der Formel wird das, was vor der Tilde `~` steht, als abhängige Variable betrachtet (hier y), und was danach steht, sind unabhängige Variablen (hier nur x). Sofern nicht ausdrücklich anders gefordert, beinhaltet das Modell auch die y-Achsenabschnitt (auch: Schnittpunkt mit der y-Achse) \beta_0.

Zum Beispiel kann die Regression der Energiedaten wie folgt durchgeführt werden:

```
import statsmodels.formula.api as smf

smfols = smf.ols("ES_Lab ~ TMK", egywth)
```

Diese Zeile konfiguriert auch wieder nur das lineare Regressionsmodell, lernt aber noch nicht die Parameter. Hierfür muss auch die Methode `.fit()` aufgerufen werden, allerdings ohne Daten und Formel, da die ja schon definiert sind.

```
smflm = smfols.fit() # Der Aufruf gibt das eigentliche Modell zurück
```

Die Parameter lassen sich hier am Attribut `params` auslesen oder mit der Methode `summary()` inklusiver Fehlerstatistiken strukturiert anzeigen. Auf diese gehen wir später noch ein.

```
smflm.params
```

```
Intercept    7.778395
TMK          3.185322
dtype: float64
```

```
smflm.summary()
```

OLS Regression Results

Dep. Variable:	ES_Lab	**R-squared:**	0.381
Model:	OLS	**Adj. R-squared:**	0.380
Method:	Least Squares	**F-statistic:**	663.7
Date:	Sat, 06 Jun 2026	**Prob (F-statistic):**	1.79e-114
Time:	11:23:19	**Log-Likelihood:**	-5107.2
No. Observations:	1082	**AIC:**	1.022e+04
Df Residuals:	1080	**BIC:**	1.023e+04
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	7.7784	1.540	5.050	0.000	4.756	10.801
TMK	3.1853	0.124	25.762	0.000	2.943	3.428

Omnibus:	61.376	**Durbin-Watson:**	0.603
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	68.207
Skew:	0.596	**Prob(JB):**	1.55e-15
Kurtosis:	2.696	**Cond. No.**	23.3

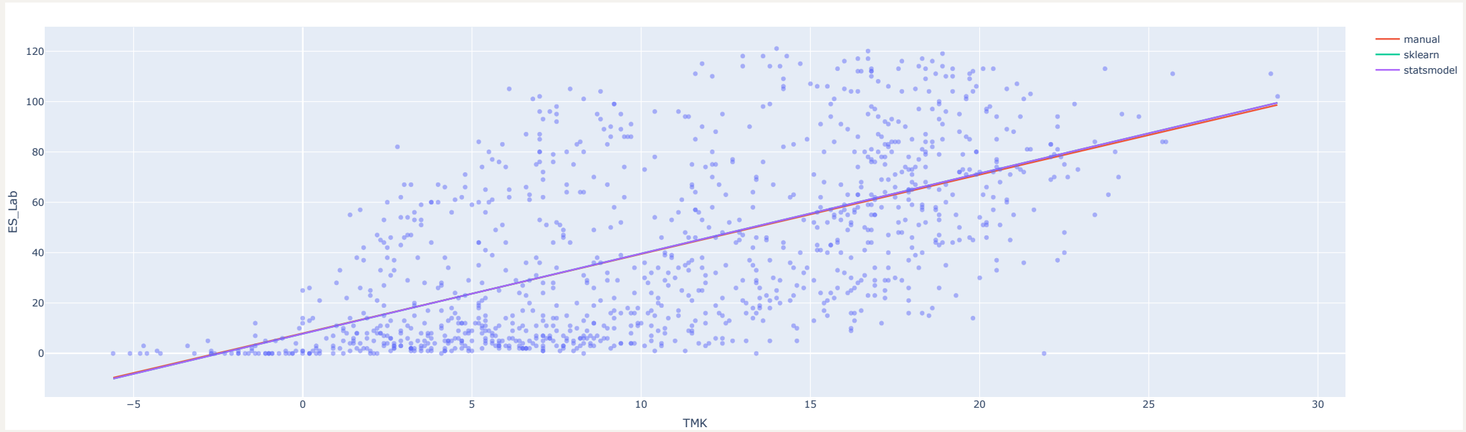
Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Eine Vorhersage machen wir hier wieder mit der Methode `predict()`. Hier ist zu beachten, dass die Methode jetzt einen DataFrame in der gleichen Gestalt, wie beim Training erwartet, also einen mit der Spalte "TMK".

```
pred_x_df  = pd.DataFrame({"TMK":pred_x})
pred_y_smf = smflm.predict(pred_x_df)
```

Wenn wir das Ergebnis darstellen, sehen wir, dass das Modell identisch ist mit dem von SciKit-Learn.

```
fig.add_scatter(x=pred_x, y=pred_y_smf, name="statsmodel")
fig
```



MULTIVARIATE LINEARE REGRESSION

Interessant wird die lineare Regression, wenn man nicht nur Modelle trainiert, die von einer Variable abhängen, sondern von mehreren Eingangsvariablen. Methodisch ist das nur eine Verallgemeinerung des bereits bekannten Modells in dem man die Beziehung zwischen einer abhängigen Zielvariable $\boldsymbol y$ und mehreren unabhängigen Eingangsvariablen $\boldsymbol x_1, \boldsymbol x_2, \ldots, \boldsymbol x_k$ modelliert. Sie wird häufig in maschinellen Lernanwendungen eingesetzt, um Vorhersagen zu treffen oder Muster in Daten zu erkennen.

Hierfür erweitern wir die ursprüngliche univariate Gleichung der linearen Regression einfach um weitere Anstiege $\beta_1, \ldots, \beta_k$ für jede unabhängige Eingangsvariable $\boldsymbol x_1, \boldsymbol x_2, \ldots, \boldsymbol x_k$:

$$\boldsymbol y = \beta_0 + \beta_1 \boldsymbol x_1 + \beta_2 \boldsymbol x_2 + \ldots + \beta_k \boldsymbol x_k + \epsilon = \boldsymbol X \boldsymbol \beta + \epsilon$$

Um die Parameter $\beta_1, \beta_2, \ldots, \beta_k$ zu schätzen, verwenden wir wieder die Methode der kleinsten Quadrate (OLS) und minimiert die Summe der quadrierten Fehler:

$$Q = \sum_{i=1}^n (y_i - (\beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + ... + \beta_k x_{i,k}))^2$$

wobei n die Anzahl der Datenpunkte ist und $y_i ,x_{i,1},x_{i,2},\ldots,x_{i,k}$ die Werte der Variablen für den i-ten Datenpunkt sind.

Um die Parameter $\beta_0, \ldots, \beta_k$ zu schätzen bestimmen wir wieder die Nullstellen der partiellen Ableitungen $\frac{\partial Q}{\partial \beta_0} =0$ bis $\frac{\partial Q}{\partial \beta_i} =0$ für $i=1,2,\ldots,k$.

Die Lösung des Minimierungsproblems ergibt die folgenden Gleichungen für die Regressionskoeffizienten:

$$\beta_0 = \frac{\sum_{i=1}^n (y_i - \bar y) \bar x_i}{\sum_{i=1}^n \bar x_i^2}$$

$$\beta_j = \frac{\sum_{i=1}^n (y_i - \bar y) (x_{i,j} - \bar x_j)}{\sum_{i=1}^n (x_{i,j} - \bar x_j)^2}$$

wobei:

- $\bar y$ der Mittelwert der abhängigen Variable ist
- $\bar x_j$ der Mittelwert der j-ten unabhängigen Variable ist
- $x_{i,j}$ der Wert der j-ten unabhängigen Variable für den i-ten Datensatzpunkt ist

Das lässt sich auch numerisch durch Matrizeninversion lösen

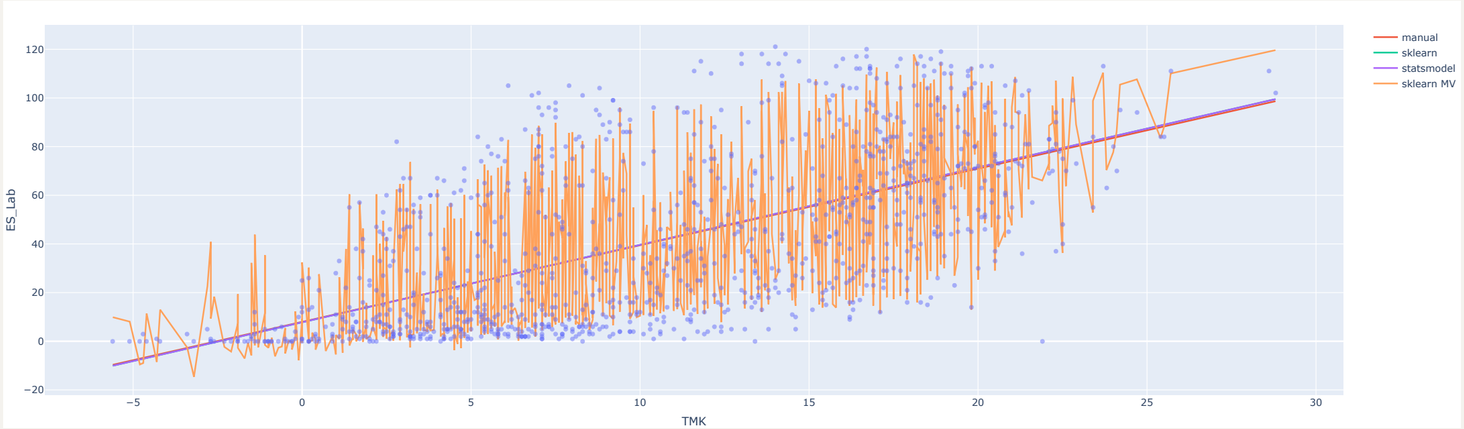
$$\boldsymbol \beta = (\boldsymbol X^T \cdot \boldsymbol X)^{-1} \cdot (\boldsymbol X^T \cdot \boldsymbol y)$$

SciKit-Learn

Wir können in SciKit-Learn sehr einfach ein multivariates Modell erstellen, indem wir eine Matrix mit allen Eingangspalten übergeben.

```
sklm_mv = LinearRegression()

egywthNoNA_MV = egywth[["SDK", "NM", "VPM", "TMK", "ES_Lab"]].dropna().sort_values("TMK") # Drop NA rows

# Modell trainieren
X_MV = egywthNoNA_MV.iloc[:,:-1].values # alles außer letzte Spalte
Y_MV = egywthNoNA_MV.ES_Lab.values
sklm_mv.fit(X_MV, Y_MV)
```

```
egywthNoNA_MV["pred"] = sklm_mv.predict(X_MV)
fig.add_scatter(x=egywthNoNA_MV.TMK, y=egywthNoNA_MV.pred, name="sklearn MV")
fig
```



Wir sehen hier, dass die neue multivariate Vorhersage zumindest in dieser Darstellung des Streudiagramms schlechter scheint. Wir schauen später, ob diese Beobachtung stimmt.

Statsmodels

Auch in Statsmodels können wir einfach ein multivariates Modell erzeugen, indem wir einfach mehr Variablen in der Formel aufnehmen. Im Falle von mehr unabhängigen Variablen können diese mit einem Pluszeichen hinzugefügt werden, zum Beispiel `"y ~ x1 + x2 + x3"`. Dies entspricht dem Regressionsmodell.

$$\boldsymbol y = \beta_0 + \beta_1 \boldsymbol x_1 + \beta_2 \boldsymbol x_2 + \beta_3 \boldsymbol x_3 + \epsilon$$

Daher bedeutet das Pluszeichen in der Formel nicht "addieren", sondern "in das Modell aufnehmen". Die Formel sollte also so gelesen werden: "Verwende $\boldsymbol y$ als abhängige Variable und nimm $\boldsymbol x_1$, $\boldsymbol x_2$ und $\boldsymbol x_3$ als erklärende Variablen auf."

Zum Beispiel können wir den Energieverbrauch unter zu Hilfenahme der Sonnenscheindauer (SDK), dem mittleren Bedeckungsgrade (NM) und des Dampfdruckes (VPM) bestimmen:

```
smflm_mv = smf.ols("ES_Lab ~ TMK + SDK + NM + VPM", data=egywth).fit()
```

```
smflm_mv.params
```

```
Intercept   -20.611237
TMK           1.997753
SDK           6.936602
NM            3.690851
VPM          -1.571116
dtype: float64
```

```
smflm_mv.summary()
```

OLS Regression Results

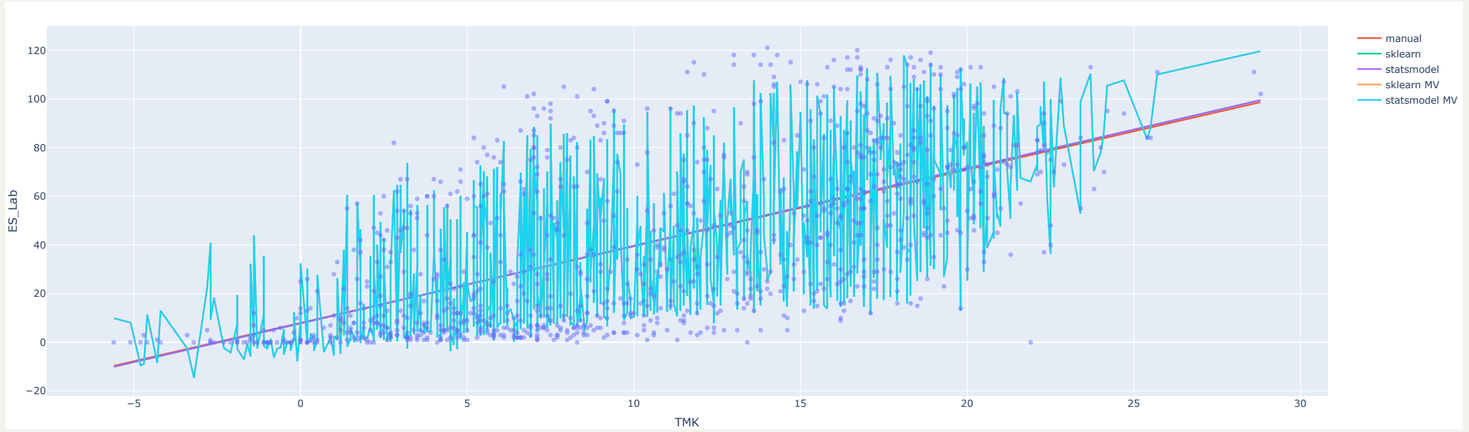
Dep. Variable:	ES_Lab	**R-squared:**	0.872
Model:	OLS	**Adj. R-squared:**	0.871
Method:	Least Squares	**F-statistic:**	1754.
Date:	Sat, 06 Jun 2026	**Prob (F-statistic):**	0.00
Time:	11:23:19	**Log-Likelihood:**	-4088.9
No. Observations:	1039	**AIC:**	8188.
Df Residuals:	1034	**BIC:**	8212.
Df Model:	4		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-20.6112	2.932	-7.030	0.000	-26.364	-14.858
TMK	1.9978	0.202	9.884	0.000	1.601	2.394
SDK	6.9366	0.180	38.489	0.000	6.583	7.290
NM	3.6909	0.336	10.986	0.000	3.032	4.350
VPM	-1.5711	0.295	-5.333	0.000	-2.149	-0.993

Omnibus:	72.688	**Durbin-Watson:**	1.079
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	184.271
Skew:	-0.378	**Prob(JB):**	9.68e-41
Kurtosis:	4.920	**Cond. No.**	140.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

```
egywthNoNA_MV["pred2"] = smflm_mv.predict(egywthNoNA_MV)
fig.add_scatter(x=egywthNoNA_MV.TMK, y=egywthNoNA_MV.pred2, name="statsmodel MV")
fig
```



Umgang mit kategorischen Eingangsvariablen und Kombinationen

Neben der bloßen Spezifizierung der Variablen bieten die Formeln viele weitere Optionen. So können wir auch kategorische Eingangsvariablen einfach zum Modell hinzufügen. Erstens ist zu beachten, dass wenn die Variable kategorisch ist (d.h. ein String), wird sie automatisch in entsprechende Dummy-Variablen umgewandelt. Fügen wir zu unserem Datensatz den Wochentag als kategorische Variable hinzu

```
egywth["Weekday"] = egywth["Date"].dt.day_name()
egywth["Weekday"]
```

```
0       Wednesday
1        Thursday
2          Friday
3        Saturday
4          Sunday
           ...
1093     Thursday
1094       Friday
1095     Saturday
1096       Sunday
1097       Monday
Name: Weekday, Length: 1098, dtype: str
```

Trainieren wir darauf ein Modell können wir die kategorische Variable direkt mit angeben.

```
smflm_mv_wd = smf.ols("ES_Lab ~ TMK + SDK + NM + VPM + Weekday", data=egywth).fit()
```

```
smflm_mv_wd.summary()
```

OLS Regression Results

Dep. Variable:	ES_Lab	**R-squared:**	0.872
Model:	OLS	**Adj. R-squared:**	0.871
Method:	Least Squares	**F-statistic:**	699.5
Date:	Sat, 06 Jun 2026	**Prob (F-statistic):**	0.00
Time:	11:23:19	**Log-Likelihood:**	-4087.5
No. Observations:	1039	**AIC:**	8197.
Df Residuals:	1028	**BIC:**	8251.
Df Model:	10		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-20.8895	3.053	-6.842	0.000	-26.881	-14.898
Weekday[T.Monday]	0.0295	1.446	0.020	0.984	-2.809	2.868
Weekday[T.Saturday]	-0.7237	1.443	-0.502	0.616	-3.555	2.107
Weekday[T.Sunday]	0.7229	1.449	0.499	0.618	-2.120	3.566
Weekday[T.Thursday]	-0.2152	1.442	-0.149	0.881	-3.045	2.615
Weekday[T.Tuesday]	1.3861	1.442	0.961	0.337	-1.444	4.216
Weekday[T.Wednesday]	0.4161	1.445	0.288	0.773	-2.419	3.251
TMK	2.0007	0.203	9.874	0.000	1.603	2.398
SDK	6.9364	0.181	38.341	0.000	6.581	7.291
NM	3.6994	0.337	10.962	0.000	3.037	4.362
VPM	-1.5740	0.295	-5.328	0.000	-2.154	-0.994

Omnibus:	70.515	**Durbin-Watson:**	1.075
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	176.111
Skew:	-0.370	**Prob(JB):**	5.73e-39
Kurtosis:	4.877	**Cond. No.**	156.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Die Zeile `Weekday[T.Monday]` zeigt den Effekt der Dummy-Variable mit dem Wert *Monday*. Darauf folgt *Saturday* da die Kategorien alphabetisch geordnet werden. Daraus folgt ein lineares Modell in der Form

$$\boldsymbol y = \beta_0 + \beta_\text{T.Monday} \boldsymbol x_\text{T.Monday} + \beta_\text{T.Saturday} \boldsymbol x_\text{T.Saturday} + \ldots + \beta_\text{T.Wednesday} \boldsymbol x_\text{T.Wednesday} + \beta_\text{TMK} \boldsymbol x_\text{TMK} + \ldots + \beta_\text{VPM} \boldsymbol x_\text{VPM} + \varepsilon$$

mit den binären Dummy-Variablen $\boldsymbol x_{T.Monday}$ für den Wochentag. Sie ist nur an Montagen 1 und an allen anderen Wochentagen 0. Das bedeutet, dass für jeden Wochentag ein spezifischer Intersept $\beta_{T.Monday}, \ldots, \beta_{T.Sunday}$ im Modell aktiviert wird.

In manchen Fällen sind die Kategorien keine Zeichenketten, sondern haben numerische Labels. Hier kann das Modell diese numerischen Kategorien nicht von numerischen Werten unterscheiden und würde einen linearen Koeffizienten hinzufügen, anstatt eines binären. Zum Beispiel können wir die Wochentage auch numerisch codieren, für das Modell ist aber nicht die Reihenfolge relevant, sondern welcher Wochentag es ist. Dies können wir erzwingen, indem wir die Transformationsfunktion "C(Variable)" mit angeben.

```
egywth["WeekdayN"] = egywth["Date"].dt.dayofweek
egywth["WeekdayN"]
```

```
0       2
1       3
2       4
3       5
4       6
       ..
1093    3
1094    4
1095    5
1096    6
1097    0
Name: WeekdayN, Length: 1098, dtype: int32
```

```
smflm_mv_wd = smf.ols("ES_Lab ~ TMK + SDK + NM + VPM + C(WeekdayN)", data=egywth).fit()
```

```
smflm_mv_wd.summary()
```

OLS Regression Results

Dep. Variable:	ES_Lab	**R-squared:**	0.872
Model:	OLS	**Adj. R-squared:**	0.871
Method:	Least Squares	**F-statistic:**	699.5
Date:	Sat, 06 Jun 2026	**Prob (F-statistic):**	0.00
Time:	11:23:19	**Log-Likelihood:**	-4087.5
No. Observations:	1039	**AIC:**	8197.
Df Residuals:	1028	**BIC:**	8251.
Df Model:	10		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-20.8600	3.111	-6.705	0.000	-26.965	-14.755
C(WeekdayN)[T.1]	1.3566	1.445	0.939	0.348	-1.479	4.192
C(WeekdayN)[T.2]	0.3866	1.447	0.267	0.789	-2.452	3.225
C(WeekdayN)[T.3]	-0.2447	1.443	-0.170	0.865	-3.077	2.588
C(WeekdayN)[T.4]	-0.0295	1.446	-0.020	0.984	-2.868	2.809
C(WeekdayN)[T.5]	-0.7533	1.444	-0.522	0.602	-3.586	2.080
C(WeekdayN)[T.6]	0.6934	1.448	0.479	0.632	-2.148	3.535
TMK	2.0007	0.203	9.874	0.000	1.603	2.398
SDK	6.9364	0.181	38.341	0.000	6.581	7.291
NM	3.6994	0.337	10.962	0.000	3.037	4.362
VPM	-1.5740	0.295	-5.328	0.000	-2.154	-0.994

Omnibus:	70.515	**Durbin-Watson:**	1.075
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	176.111
Skew:	-0.370	**Prob(JB):**	5.73e-39
Kurtosis:	4.877	**Cond. No.**	161.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Hierbei lassen sich auch Kombinationen betrachten. Zum Beispiel mag der Energieverbrauch nicht nur von den Wochentagen abhängen, sondern nutzt auch an Heiztagen und Kühltagen unterschiedliche Betriebsstrategien. Um das auszudrücken kann man kategorische Variablen mit $*$ kombinieren, was in diesem Fall keine Multiplikationsoperator ist, sondern für die Kombinationen steht.

```
smflm_mv_wd2 = smf.ols("ES_Lab ~ TMK + SDK + NM + VPM + C(WeekdayN)*HeizKuehlTage", data=egywth).fit()
```

```
smflm_mv_wd2.summary()
```

OLS Regression Results

Dep. Variable:	ES_Lab	**R-squared:**	0.876
Model:	OLS	**Adj. R-squared:**	0.873
Method:	Least Squares	**F-statistic:**	298.8
Date:	Sat, 06 Jun 2026	**Prob (F-statistic):**	0.00
Time:	11:23:19	**Log-Likelihood:**	-4070.0
No. Observations:	1039	**AIC:**	8190.
Df Residuals:	1014	**BIC:**	8314.
Df Model:	24		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-18.3083	3.315	-5.523	0.000	-24.813	-11.803
C(WeekdayN)[T.1]	2.1577	1.754	1.230	0.219	-1.284	5.600
C(WeekdayN)[T.2]	1.0729	1.769	0.606	0.544	-2.398	4.544
C(WeekdayN)[T.3]	0.3539	1.752	0.202	0.840	-3.084	3.791
C(WeekdayN)[T.4]	0.2426	1.761	0.138	0.890	-3.213	3.698
C(WeekdayN)[T.5]	-0.3289	1.747	-0.188	0.851	-3.758	3.100
C(WeekdayN)[T.6]	-0.3372	1.755	-0.192	0.848	-3.781	3.106
HeizKuehlTage[T.Kühlgradtag]	3.9698	5.915	0.671	0.502	-7.637	15.577
HeizKuehlTage[T.Normaltag]	5.2019	2.512	2.071	0.039	0.272	10.132
C(WeekdayN)[T.1]:HeizKuehlTage[T.Kühlgradtag]	-3.4876	9.169	-0.380	0.704	-21.480	14.505
C(WeekdayN)[T.2]:HeizKuehlTage[T.Kühlgradtag]	-18.8997	9.195	-2.055	0.040	-36.943	-0.856
C(WeekdayN)[T.3]:HeizKuehlTage[T.Kühlgradtag]	-5.9965	9.175	-0.654	0.514	-24.001	12.008
C(WeekdayN)[T.4]:HeizKuehlTage[T.Kühlgradtag]	-15.5864	8.474	-1.839	0.066	-32.214	1.042
C(WeekdayN)[T.5]:HeizKuehlTage[T.Kühlgradtag]	-4.4747	7.990	-0.560	0.576	-20.154	11.204
C(WeekdayN)[T.6]:HeizKuehlTage[T.Kühlgradtag]	-0.5844	7.663	-0.076	0.939	-15.621	14.452
C(WeekdayN)[T.1]:HeizKuehlTage[T.Normaltag]	-2.1827	3.128	-0.698	0.486	-8.322	3.956

C(WeekdayN)[T.2]:HeizKuehlTage[T.Normaltag]	-1.1488	3.096	-0.371	0.711	-7.225	4.927
C(WeekdayN)[T.3]:HeizKuehlTage[T.Normaltag]	-1.3698	3.121	-0.439	0.661	-7.494	4.755
C(WeekdayN)[T.4]:HeizKuehlTage[T.Normaltag]	0.6019	3.118	0.193	0.847	-5.516	6.720
C(WeekdayN)[T.5]:HeizKuehlTage[T.Normaltag]	-0.4470	3.157	-0.142	0.887	-6.643	5.749
C(WeekdayN)[T.6]:HeizKuehlTage[T.Normaltag]	4.1547	3.175	1.309	0.191	-2.075	10.385
TMK	1.9744	0.204	9.677	0.000	1.574	2.375
SDK	6.8849	0.182	37.827	0.000	6.528	7.242
NM	3.6449	0.336	10.851	0.000	2.986	4.304
VPM	-1.9031	0.309	-6.158	0.000	-2.510	-1.297

Omnibus:	64.597	Durbin-Watson:	1.101
Prob(Omnibus):	0.000	Jarque-Bera (JB):	184.425
Skew:	-0.282	Prob(JB):	8.97e-41
Kurtosis:	4.985	Cond. No.	777.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

MODELLQUALITÄT

Residuen

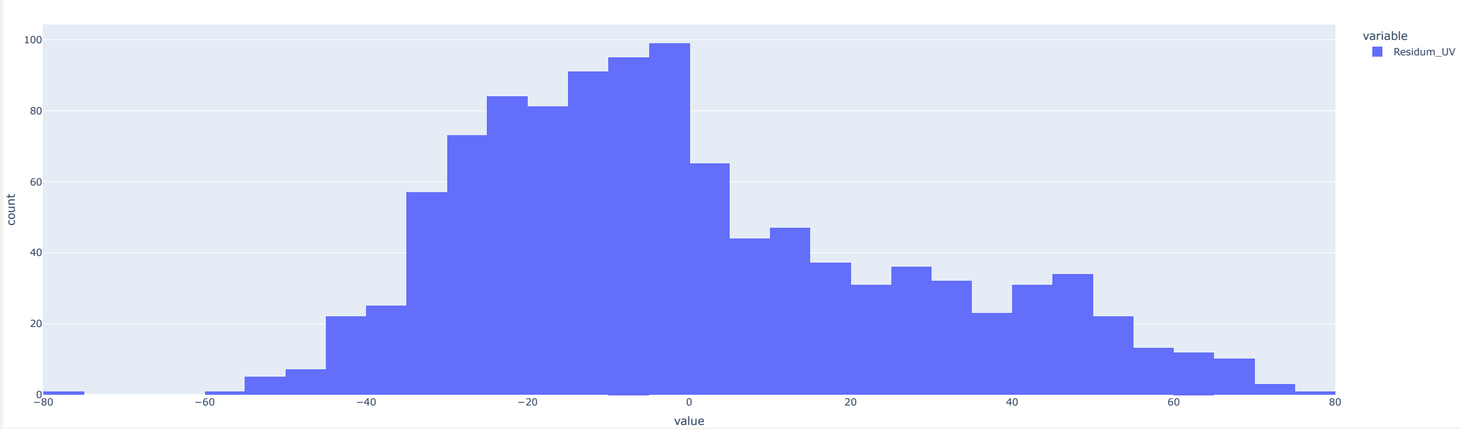
Ein wichtiger Aspekt im Maschinellen Lernen ist die Bewertung der Qualität von Modellen, um auf dieser Basis zu entscheiden, ob die Fehler im Rahmen sind und welche Modelle geeignet sind. Die Grundlage für diese Bewertung sind meist die *Residuen* oder der Vorhersagefehler, die wir bereits zuvor kennen gelernt haben.

Betrachten wir einmal die Residuen für unser Beispiel von dem univariaten und multivariaten Linearen Modellen. Wir berechnen es, indem wir die Vorhersage vom Zielwert subtrahieren.

```
egywth["Pred_UV"]=smflm.predict(egywth)
egywth["Residum_UV"]= egywth.ES_Lab - egywth.Pred_UV
```

Wie bereits diskutiert, basiert die Annahme des Linearen Modells darauf, dass dieser Fehler Normalverteilt ist. Mit einem Histogramm können wir uns die Verteilung anschauen.

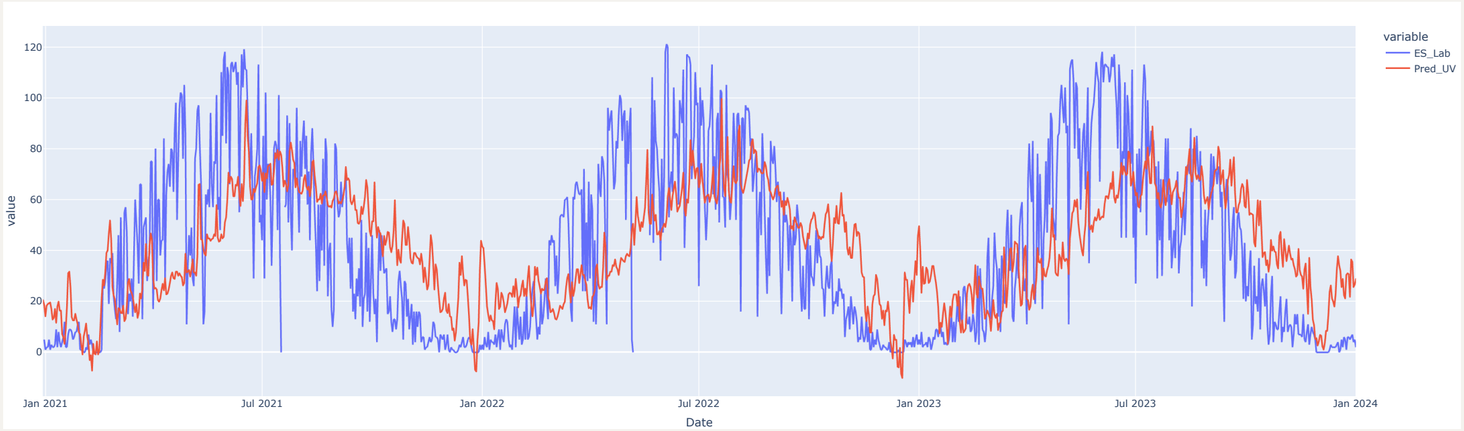
```
px.histogram(egywth.Residum_UV)
```



Die Verteilung des Residuums ist nicht Normalverteilt, sondern linkslastig. Das ist bereits ein Indiz, dass das Model vielleicht nicht optimal ist.

Schauen wir uns statt dem Streudiagramm, die Vorhersage in einem Liniendiagramm an.

```
px.line(egywth, x="Date", y=["ES_Lab", "Pred_UV"])
```

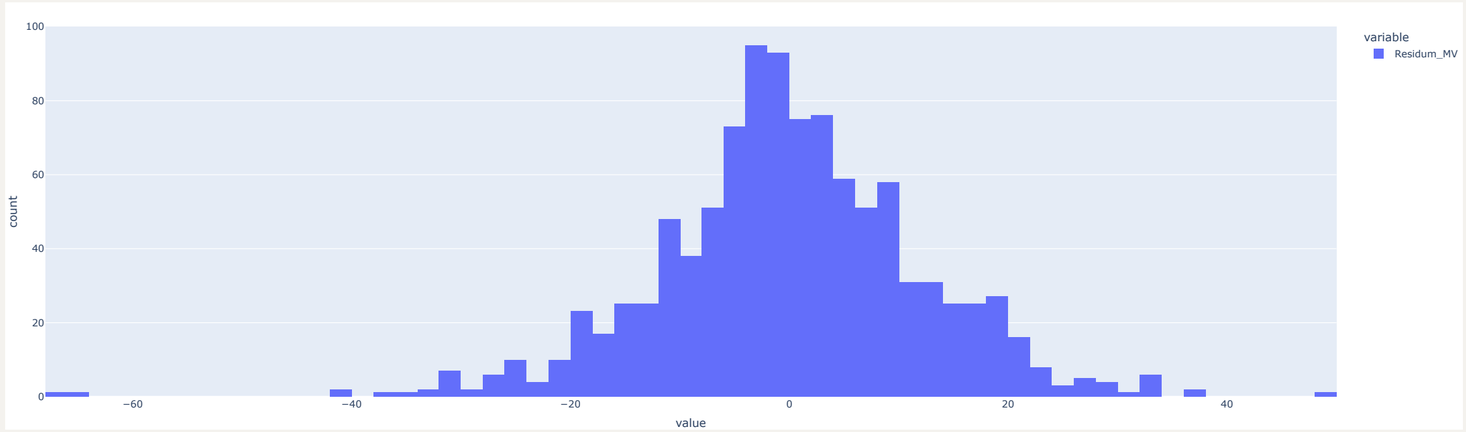


Jetzt sehen wir, dass das ursprüngliche lineare Modell im Streudiagramm recht gut aussah, weil es den mittleren Trend gut erfasst hat. In der Zeitreihe offenbart sich allerdings eine recht klare Abweichung vom Original.

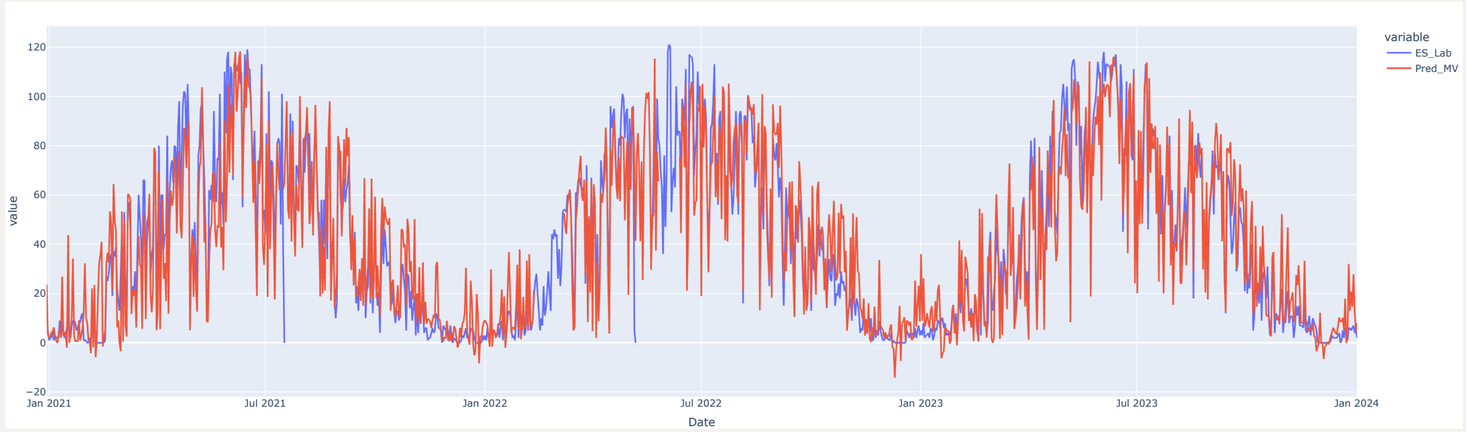
Vergleichen wir das mit dem multivariaten Modell

```
egywth["Pred_MV"]=smflm_mv_wd2.predict(egywth)
egywth["Residum_MV"]= egywth.ES_Lab - egywth.Pred_MV
```

```
px.histogram(egywth.Residum_MV)
```




```
px.line(egywth, x="Date", y=["ES_Lab", "Pred_MV"])
```



Hier sehen wir, dass die Verteilung des Residuums eher Normalverteilt ist und die Grundform der Zeitreihe gut erfasst wird. Dennoch sind deutliche Abweichungen sichtbar.

Mean Square Error (MSE)

Um diesen Vergleich von Modellen professionell verwendet man unterschiedliche Kennwerte. Der erste Kennwert ist der Mean Square Error, auf Deutsch auch die mittlere quadratische Abweichung genannt, ist eine wichtige Kenngröße hierfür.

Der MSE berechnet im Wesentlichen den durchschnittlichen quadratischen Fehler zwischen den tatsächlichen Werten y_i und den von einem Modell vorhergesagten Werten $\hat y_i$.

$$\text{MSE} = \frac{1}{N} \sum_i e_i^2$$

Ein kleinerer MSE-Wert deutet auf ein besseres Modell hin. Vergleichen wir die Werte für unsere beiden Modelle

```
egywth.Residum_UV.pow(2).mean()
```

```
np.float64(736.8486474358979)
```

```
egywth.Residum_MV.pow(2).mean()
```

```
np.float64(147.90321019931775)
```

und wir sehen, dass der MSE für das zweite Modell besser ist.

Der MSE ist in der Einheit des quadrierten Zielwertes. Daher kann die Interpretation der absoluten Größe des MSE manchmal schwierig sein. Der MSE ist empfindlich gegenüber Ausreißern in den Daten, die den Wert verfälschen können.

Root Mean Squared Error (RMSE)

Die RMSE ist die Quadratwurzel des MSE und hat somit die Einheit wie der Zielwert, was die Interpretation einfacher macht. Gleichzeitig können bestehende Interpretationen des MSE übernommen werden.

$$\text{RMSE} = \sqrt{\text{MSE}}$$

```
np.sqrt(egywth.Residum_UV.pow(2).mean())
```

```
np.float64(27.144956206188617)
```

```
np.sqrt(egywth.Residum_MV.pow(2).mean())
```

```
np.float64(12.16154637368611)
```

Mean Absolute Error (MAE)

Der MAE misst die durchschnittliche absolute Abweichung zwischen den vorhergesagten und den tatsächlichen Werten.

$$\text{MAE} = \frac{1}{N} \sum_i \left| e_i \right|$$

Der MAE ist weniger empfindlich gegenüber Ausreißern als der MSE. Die Einheit des MAE ist die gleiche wie die Einheit des Zielwertes, was die Interpretation einfacher macht.

```
egywth.Residum_UV.abs().mean()
```

```
np.float64(21.892016248549634)
```

```
egywth.Residum_MV.abs().mean()
```

```
np.float64(9.110498926081684)
```

Mean Absolute Percentage Error (MAPE)

Der MAPE misst den durchschnittlichen prozentualen absoluten Fehler. Er ist zu interpretieren, ohne den originalen Wertebereich zu kennen. Ein niedriger MAPE-Wert deutet auf eine gute Übereinstimmung zwischen den vorhergesagten und den tatsächlichen Werten hin.

$$\text{MAPE} = \frac{100}{N} \sum_i \left| \frac{e_i}{y_i} \right|$$

```
egywth.Residum_UV.abs().mean()*100
```

```
np.float64(2189.2016248549635)
```

```
egywth.Residum_MV.abs().mean()*100
```

```
np.float64(911.0498926081684)
```

Bestimmungskoeffizient R^2

Der Bestimmungskoeffizient R^2 gibt den Anteil der durch das Regressionsmodell erklärten Variabilität an der gesamten Variabilität in den Daten wieder. Bei gut passenden Modellen liegt er oft nahe bei 1. Ein Wert von 0 bedeutet, dass das Modell auf diesen Daten nicht besser ist als die Vorhersage mit dem Mittelwert. In manchen Situationen kann R^2 auch negativ werden, zum Beispiel auf Testdaten oder bei schlecht passenden Modellen.

Er bestimmt sich aus der Sum of Squared Error (SSE) und der Total Sum of Squares (TSS) zu

$$R^2 = 1 - \frac{\text{SSE}}{\text{TSS}}$$

Die Gesamtsumme der Quadrate (TSS) misst die Summe der quadrierten Abweichungen der tatsächlichen Werte (y_i) vom Mittelwert (\bar y) aller Datenpunkte. Sie repräsentiert die Gesamtvariabilität in den Daten. Er wird berechnet durch

\text{TSS} = \sum_{i=1}^{N} (y_{i} - \bar{y})^{2}.

Die Summe der quadratischen Abweichungen (SSE) misst die Summe der quadrierten Abweichungen zwischen den vorhergesagten Werten \hat y_i und den tatsächlichen Werten y_i. Sie dient zur Beurteilung der Genauigkeit des Modells durch:

\text{SSE} = \sum_{i=1}^{N} \varepsilon_i^2 = \sum_{i=1}^{N} (y_i - \hat y_i)^2.

Wir können das manuell berechnen

```
TSS = (egywth.ES_Lab - egywth.ES_Lab.mean()).pow(2).sum()
SSE = egywth.Residum_UV.pow(2).sum()
R2 = 1 - SSE/TSS
R2
```

```
np.float64(0.38061649401642605)
```

```
TSS = (egywth.ES_Lab - egywth.ES_Lab.mean()).pow(2).sum()
SSE = egywth.Residum_MV.pow(2).sum()
R2 = 1 - SSE/TSS
R2
```

```
np.float64(0.8806156958265965)
```

Oder direkt das von `statsmodel` bereitgestellte `rsquared` -Attribut nutzen

```
smflm.rsquared
```

```
np.float64(0.38061649401642617)
```

```
smflm_mv_wd2.rsquared
```

```
np.float64(0.876122441552206)
```

Hier ist wichtig, dass das Modell besser ist, wenn der R^2 Wert größer ist und nicht kleiner, wie bei den anderen Fehlerwerten.

Bei *sklearn* können wir dafür die Methode `score` nutzen.

```
sklm.score(X, Y)
```

```
0.38061649401642617
```

```
sklm_mv.score(X_MV, Y_MV)
```

```
0.8715365639979806
```

QUESTIONS