Logistische Regression

Joern Ploennigs · AI4SC

> Die kürzesten Wörter, nämlich ‚ja' und ‚nein' erfordern das meiste Nachdenken.
>
> — Pythagoras von Samos

Wir haben gesehen, dass wir linearen Regressionsmodellen nutzen können um kontinuierliche Ergebnisse vorherzusagen. Sie eignen sich allerdings nicht gut für binäre (kategorische) Ergebnisse die zwei Ausgänge wie ‚Ja' (1) oder ‚Nein' (0) haben, da die klare Trennung zwischen Ja und Nein im Training des Modells nicht gut betont wird (aufgrund der linearen Eigenschaft). Deshalb nutzt man für binäre Ergebnisse Modelle der *Logistischen Regression*. Das sind statistische Modelle, welche die Wahrscheinlichkeit schätzt, dass ein bestimmtes Ereignis eintritt oder nicht, z.B. ob ein Kunde ein Produkt kauft (Ja/Nein), ob er eine Infektion hat (Ja/Nein) oder ob ein Gerät an/aus ist. Dabei kann das Modell kontinuierliche Eingangswerte nutzen.

Modell Grundlagen

Wie bei der linearen Regression beginnt die logistische Regression mit einer linearen Kombination der Eingabevariablen (Prädiktoren). Diese lineare Kombination kann als z_i dargestellt werden:

$$z_i = \beta_0 + \beta_1 \cdot x_{1,i} + \beta_2 \cdot x_{2,i} + ... + \beta_n \cdot x_{n,i}.$$

Statt die lineare Kombination direkt als Ergebnis zu nehmen, wird sie durch die logistische Funktion (auch Sigmoid-Funktion genannt) transformiert. Die logistische Funktion ist definiert als:

$$P(y_i = 1 \mid \bar{x}_i) = \sigma_i = \frac{1}{1 + e^{-z_i}} = \frac{1}{1 + e^{-\beta_0 - \beta_1 \cdot x_{1,i} - \beta_2 \cdot x_{2,i} - ... - \beta_n \cdot x_{n,i}}}.$$

Diese Funktion nimmt beliebige reelle Zahlen und transformiert sie in den Bereich $(0, 1)$, was als Wahrscheinlichkeit interpretiert werden kann. Die Ausgabe der logistischen Funktion σ_i gibt die Wahrscheinlichkeit an, dass das Ergebnis 1 ist (z. B. Kunde kauft das Produkt) gegeben die

Eingabewerte. Die Wahrscheinlichkeit, dass das Ergebnis 0 ist, ist einfach $P(y_i = 0 \mid \bar{x}_i) = 1 - \sigma_i$.

Die lineare Kombination (z.B. aus Temperatur, Alter und Druck) wird durch die Sigmoid-Funktion in eine Wahrscheinlichkeit zwischen 0 und 1 umgewandelt: z.B. 0.84 (84 %) Wahrscheinlichkeit für Ereignis ‚Ja'.

```
Unable to display output for mime type(s): text/html
```

Die Regressionskoeffizienten $\beta_0, \beta_1, ..., \beta_n$ geben an, wie sich eine Einheitsänderung der jeweiligen unabhängigen Variablen $x_1, ..., x_n$ auf die Logarithmus der Chancen (Odds Ratio) des Ereignisses y=1 auswirkt.

- Ein positiver Regressionskoeffizient β_i bedeutet, dass eine Zunahme der Variablen x_i um eine Einheit die Chancen des Ereignisses $y_i = 1$ erhöht.
- Ein negativer Regressionskoeffizient β_i bedeutet, dass eine Zunahme der Variablen x_i um eine Einheit die Chancen des Ereignisses $y_i = 1$ verringert.
- Ein Wert von 0 des Regressionskoeffizienten β_i bedeutet, dass die Variable x_i keinen Einfluss auf die Wahrscheinlichkeit des Ereignisses $y_i = 1$ hat.

Die Interpretation der Regressionskoeffizienten in Bezug auf die Wahrscheinlichkeit des Ereignisses $y_i = 1$ erfordert die Berechnung der Chancen und deren anschließende Interpretation.

Parameteranpassung

Die Anpassung des logistischen Regressionsmodells an Daten erfolgt typischerweise durch *Maximum-Likelihood-Schätzung (MLE)*. Dies bedeutet, dass die Koeffizienten $\bar{\beta}$ so gewählt werden, dass die Wahrscheinlichkeit der beobachteten Daten maximiert wird.

Die Likelihood-Funktion für eine Menge von n Trainingsbeispielen x_i, y_i ist:

$$L(\bar{\beta}) = \prod_{i=1}^{n} P(y_i \mid \bar{x}_i) = \prod_{i=1}^{n} \sigma_i^{y_i} (1 - \sigma_i)^{1-y_i} = \prod_{i=1}^{n} \{ \begin{matrix} \sigma_i & y_i = 1, \\ 1 - \sigma_i & y_i = 0. \end{matrix}$$

Da das Produkt schwerer zu berechnen ist, nutzt man lieber die Log-Likelihood. Dafür berechnen wir den Logarithmus der Likelihood-Funktion:

$$\log L(\bar{\beta}) = \log \left(\prod_{i=1}^{n} P(y_i \mid \bar{x}_i) \right) = \sum_{i=1}^{n} (y_i \log(\sigma_i) + (1 - y_i) \log(1 - \sigma_i)).$$

Um die Koeffizienten $\bar{\beta}$ zu bestimmen, berechnet man wieder die Nullstellen der partiellen Ableitung nach $\beta_0, \beta_1, ..., \beta_n$.

$$0 = \frac{\partial \ell}{\partial \beta_0} = \sum_{k=1}^{n} (y_k - \sigma_k)$$

$$0 = \frac{\partial \ell}{\partial \beta_i} = \sum_{k=1}^{n} (y_k - \sigma_k) x_{k,i}$$

und löst das resultierende Gleichungssystem mit numerischen Methoden. Das ist notwendig, da die Log-Likelihood-Funktion nicht linear ist (sie enthält die e-Funktion). Deshalb wird zur Maximierung oft ein iteratives Verfahren wie das Newtonverfahren (Newton-Raphson-Algorithmus) verwendet. Dieser Algorithmus bestimmt die Parameter $\bar{\beta}$ iterativ:

$$\bar{\beta}_{t+1} = \bar{\beta}_t - H\left(\bar{\beta}_t\right)^{-1} \Delta \log L\left(\bar{\beta}_t\right),$$

bis eine hinreichende Genauigkeit erzielt wird. Hier ist $\Delta \log L\left(\bar{\beta}_t\right)$ der Gradient (Vektor der partiellen Ableitungen) und $H\left(\bar{\beta}_t\right)$ die Hessian-Matrix (Matrix der zweiten Ableitungen) der Log-Likelihood-Funktion in Bezug auf die Parameter $\bar{\beta}$ mit

$$\Delta \log L\left(\bar{\beta}_t\right) = \sum_{k=1}^{n} (y_k - \sigma_k) \bar{x}_k$$

$$H\left(\bar{\beta}_t\right) = -\sum_{k=1}^{n} \sigma_k (1 - \sigma_k) \bar{x}_k \bar{x}_k^T$$

Logistische Regression in Python

Wie bei der linearen Regression können wir sowohl die Bibliotheken *statsmodels* als auch *sklearn* auch für die logistische Regression verwenden. Die Verwendung ist ähnlich wie bei der linearen Regression, aber beide Bibliotheken haben ihre eigenen Eigenheiten.

Als Beispiel nutzen wir wieder den Energiedatensatz der Universität Rostock. Wir wollen hier vorhersagen, wann die Anlagen in „HT 740“ aktiviert sind, also Energie verbrauchen.

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

egywth = pd.read_csv("../data/UROS/Energy1D_weather_clean.csv",
parse_dates=[0])
egywth["Weekday"] = egywth["Date"].dt.day_name()
egywth["EV_HT_740_IS_ON"] = egywth.EV_HT_740==0
egywth.EV_HT_740_IS_ON.value_counts()
```

```
EV_HT_740_IS_ON
False    553
True     545
Name: count, dtype: int64
```

Logistische Regression in Python mit der Statsmodels Formel API

Bei der Verwendung von *statsmodels* müssen wir eine ähnliche Formel wie bei der linearen Regression angeben. Bevor wir das jedoch tun können, müssen wir eine Eigenheit in statsmodels ansprechen - nämlich, dass es nicht erlaubt ist, dass das Ergebnis eine logische oder Zeichenfolgenvariable ist. Es *muss* eine 0/1 (numerische) Variable sein, ansonsten tritt ein Fehler auf:

> ValueError: wthegy hat sich zu einem Array mit mehreren Spalten ausgewertet, das die Form (2675, 2) hat. Dies tritt auf, wenn die Variable, die in wthegy umgewandelt wurde, nicht numerisch ist (z. B. bool oder str).

Um also ein Modell zu erstellen, bei dem wir die Wahrscheinlichkeit der Behandlung schätzen, müssen wir zuerst eine neue Variable erstellen, die nur die numerische Version von `EV_HT_740_IS_ON` ist. Hierfür wandeln wir sie in ein Integer.

```
egywth["EV_HT_740_ON"] = egywth.EV_HT_740_IS_ON.astype(int)
egywth.EV_HT_740_ON.value_counts()
```

```
EV_HT_740_ON
0    553
1    545
Name: count, dtype: int64
```

Wir sehen, dass beide Werte relativ gleich verteilt sind. Mit dieser numerischen Variable können wir jetzt ein logistische Regressionsmodell erstellen. Hierfür geben wir wie beim linearen Modell eine Formel der Eingangsparameter an, die kontinuierlich sein können als auch kategorisch wie der Wochentag

```
import statsmodels.formula.api as smf
smm0 = smf.logit("EV_HT_740_ON ~ TMK + SDK + NM + VPM + Weekday", data=egywth)
smm  = smm0.fit()
```

```
Optimization terminated successfully.
         Current function value: 0.594367
         Iterations 8
```

Dieser Code ähnelt sehr der linearen Regression, der einzige Unterschied besteht darin, dass `smf.logit` anstelle von `smf.ols` verwendet wird. Genau wie bei der linearen Regression verwendet die logistische Regression R-Style-Formeln. Daher ist das spezifische Modell, das wir verwenden wobei

$$\boldsymbol{z} = \beta_0 + \beta_{\text{T.Monday}}\boldsymbol{x}_{\text{T.Monday}} + \beta_{\text{T.Saturday}}\boldsymbol{x}_{\text{T.Saturday}} + \ldots + \beta_{\text{T.Wednesday}}\boldsymbol{x}_{\text{T.Wednesday}} + \beta_{\text{TMK}}\boldsymbol{x}_{\text{TMK}} + \ldots + \beta_{\text{VPM}}\boldsymbol{x}_{\text{VPM}}.$$

Wir können die geschätzten Werte mit anzeigen.

```
smm.params
```

```
Intercept               0.123141
Weekday[T.Monday]       0.054098
Weekday[T.Saturday]     0.087200
Weekday[T.Sunday]       4.196551
Weekday[T.Thursday]    -0.015990
Weekday[T.Tuesday]     -0.085271
Weekday[T.Wednesday]   -0.058213
TMK                    -0.000715
SDK                    -0.043613
NM                     -0.001606
VPM                    -0.014591
dtype: float64
```

Wir können auch eine vollständige Zusammenfassung der Ausgabe erhalten.

```
smm.summary()
```

Die Ausgabe ähnelt der linearen Regression, weist jedoch bestimmte Kennzahlen für die Modellqualität von logistischer Regression.

```
egywth["flm_P"] = smm.predict(egywth)
egywth["flm_T"] = egywth.flm_P.map(lambda x: 0 if x < 0.5 else 1,
na_action='ignore')
px.line(egywth, x="Date", y=["EV_HT_740_ON", "flm_T"])
```

```
Unable to display output for mime type(s): text/html
```

In dem Linien-Diagram sieht man die Übereinstimmung nicht wirklich gut. Für binäre Werte eignen sich eher Heatmaps.

```
px.imshow(egywth[["EV_HT_740_ON", "flm_T"]].T, x=egywth["Date"],
color_continuous_scale='Viridis')
```

```
Unable to display output for mime type(s): text/html
```

Hier zeigt sich eine gute Übereinstimmung bis Mitte 2022 aber dann falsche Vorhersagen für den Zeitraum danach.

Logistische Regression mit Scikit-learn

Die Durchführung einer logistischen Analyse mit *sklearn* ähnelt in vieler Hinsicht der Durchführung einer linearen Regression in *sklearn*. Die Tatsache, dass wir den gleichen Ansatz mit logistischer Regression verwenden können wie im Fall der linearen Regression, ist ein großer

Vorteil von *sklearn*: Der gleiche Ansatz gilt auch für andere Modelle, daher ist es sehr einfach, mit verschiedenen Modellen zu experimentieren. Sehr wenig zusätzlicher Code und keine zusätzliche Datenverarbeitung sind erforderlich.

Zuerst importieren und richten wir das Modell ein:

```
# we need to import the model
from sklearn.linear_model import LogisticRegression
# create the model
sklm = LogisticRegression()
```

Wir können verschiedene Optionen für `LogisticRegression` angeben, aber im Moment sind wir mit den Standardwerten zufrieden.

Als nächstes erstellen wir die Designmatrix, wir müssen nur die „age“-Variable extrahieren und sicherstellen, dass das Ergebnis ein Datenrahmen (oder eine Matrix) ist:

```
egywthNoNA = egywth[["SDK", "NM", "VPM", "TMK", "EV_HT_740_ON"]].dropna() #
Drop NA rows

# Modell trainieren
X = egywthNoNA.iloc[:,:-1].values # alles außer letzte Spalte
Y = egywthNoNA.EV_HT_740_ON.values
sklm.fit(X, Y)
```

	penalty penalty: {'l1', 'l2', 'elasticnet', None}, default='l2' Specify the norm of the penalty: - `None`: no penalty is added; - `'l2'`: add a L2 penalty term and it is the default choice; - `'l1'`: add a L1 penalty term; - `'elasticnet'`: both L1 and L2 penalty terms are added. .. warning:: Some penalties may not work with some solvers. See the parameter `solver` below, to know the compatibility between the pe-	'deprecated'

	nalty and solver. .. versionadded:: 0.19 l1 penalty with SAGA solver (allowing 'multinomial' + L1) .. deprecated:: 1.8 `penalty` was deprecated in version 1.8 and will be removed in 1.10. Use `l1_ratio` instead. `l1_ratio=0` for `penalty='l2'`, `l1_ratio=1` for `penalty='l1'` and `l1_ratio` set to any float between 0 and 1 for `'penalty='elasticnet'`.	
	C C: float, default=1.0 Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization. `C=np.inf` results in unpenalized logistic regression. For a visual example on the effect of tuning the `C` parameter with an L1 penalty, see: :ref:`sphx_glr_auto_examples_linear_model_plot_logistic_path.py`.	1.0
	l1_ratio l1_ratio: float, default=0.0 The Elastic-Net mixing parameter, with `0 <= l1_ratio <= 1`. Setting `l1_ratio=1` gives a pure L1-	0.0

	penalty, setting `l1_ratio=0` a pure L2-penalty. Any value between 0 and 1 gives an Elastic-Net penalty of the form `l1_ratio * L1 + (1 - l1_ratio) * L2`. .. warning:: Certain values of `l1_ratio`, i.e. some penalties, may not work with some solvers. See the parameter `solver` below, to know the compatibility between the penalty and solver. .. versionchanged:: 1.8 Default value changed from None to 0.0. .. deprecated:: 1.8 `None` is deprecated and will be removed in version 1.10. Always use `l1_ratio` to specify the penalty type.	
	dual dual: bool, default=False Dual (constrained) or primal (regularized, see also :ref:`this equation `) formulation. Dual formulation is only implemented for l2 penalty with liblinear solver. Prefer `dual=False` when n_samples > n_features.	False
	tol tol: float, default=1e-4	0.0001

	Tolerance for stopping criteria.	
	fit_intercept fit_intercept: bool, default=True Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.	True
	intercept_scaling intercept_scaling: float, default=1 Useful only when the solver `liblinear` is used and `self.fit_intercept` is set to `True`. In this case, `x` becomes `[x, self.intercept_scaling]`, i.e. a "synthetic" feature with constant value equal to `intercept_scaling` is appended to the instance vector. The intercept becomes ``intercept_scaling * synthetic_feature_weight``. .. note:: The synthetic feature weight is subject to L1 or L2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) `intercept_scaling` has to be increased.	1
	class_weight class_weight: dict or 'balanced', default=None	None

	Weights associated with classes in the form ``{class_label: weight}``. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as ``n_samples / (n_classes * np.bincount(y))``. Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified. .. versionadded:: 0.17 *class_weight='balanced'*	
	random_state random_state: int, RandomState instance, default=None Used when ``solver`` == 'sag', 'saga' or 'liblinear' to shuffle the data. See :term:`Glossary ` for details.	None
	solver solver: {'lbfgs', 'liblinear', 'newton-cg', 'newton-cholesky', 'sag', 'saga'}, default='lbfgs' Algorithm to use in the optimization problem. Default is 'lbfgs'.	'lbfgs'

	To choose a solver, you might want to consider the following aspects: - 'lbfgs' is a good default solver because it works reasonably well for a wide class of problems. - For :term:`multiclass` problems (`n_classes >= 3`), all solvers except 'liblinear' minimize the full multinomial loss, 'liblinear' will raise an error. - 'newton-cholesky' is a good choice for `n_samples` >> `n_features * n_classes`, especially with one-hot encoded categorical features with rare categories. Be aware that the memory usage of this solver has a quadratic dependency on `n_features * n_classes` because it explicitly computes the full Hessian matrix. - For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones; - 'liblinear' can only handle binary classification by default. To apply a one-versus-rest scheme for the multiclass setting one can wrap it with the :class:`~sklearn.multiclass.OneVsRestClassifier`. .. warning:: The choice of the algorithm	

	depends on the penalty chosen (`l1_ratio=0` for L2-penalty, `l1_ratio=1` for L1-penalty and `0 < l1_ratio < 1` for Elastic-Net) and on (multinomial) multiclass support: ========== ========== ========== solver l1_ratio multinomial multiclass ========== ========== ========== 'lbfgs' l1_ratio=0 yes 'liblinear' l1_ratio=1 or l1_ratio=0 no 'newton-cg' l1_ratio=0 yes 'newton-cholesky' l1_ratio=0 yes 'sag' l1_ratio=0 yes 'saga' 0<=l1_ratio<=1 yes ========== ========== ========== .. note:: 'sag' and 'saga' fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from :mod:`sklearn.preprocessing`. .. seealso:: Refer to the :ref:`User Guide ` for more information regarding :class:`LogisticRegression` and more specifically the :ref:`Table ` summarizing solver/penalty	

	supports. .. versionadded:: 0.17 Stochastic Average Gradient (SAG) descent solver. Multinomial support in version 0.18. .. versionadded:: 0.19 SAGA solver. .. versionchanged:: 0.22 The default solver changed from 'liblinear' to 'lbfgs' in 0.22. .. versionadded:: 1.2 newton-cholesky solver. Multinomial support in version 1.6.	
	max_iter max_iter: int, default=100 Maximum number of iterations taken for the solvers to converge.	100
	verbose verbose: int, default=0 For the liblinear and lbfgs solvers set verbose to any positive number for verbosity.	0
	warm_start warm_start: bool, default=False When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. Useless for liblinear solver. See :term:`the Glossary `.	False

	.. versionadded:: 0.17 *warm_start* to support *lbfgs*, *newton-cg*, *sag*, *saga* solvers.	
	n_jobs n_jobs: int, default=None Does not have any effect. .. deprecated:: 1.8 `n_jobs` is deprecated in version 1.8 and will be removed in 1.10.	None

Wir können die gelernten Parameter wieder anschauen mit

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

```
(array([0.26382612]),
 array([[-0.0285415 ,  0.01756992, -0.01759732,  0.00113189]]))
```

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

```
egywthNoNA["flm_P"] = sklm.predict_proba(X)[:, 1]
egywthNoNA["flm_T"] = egywthNoNA.flm_P.map(lambda x: 0 if x < 0.5 else 1,
na_action='ignore')
```

```
px.imshow(egywthNoNA[["EV_HT_740_ON", "flm_T"]].T, x=egywthNoNA.index,
color_continuous_scale='Viridis')
```

```
Unable to display output for mime type(s): text/html
```

Hier zeigt sich, dass das Modell im hinteren Bereich nach 2022 besser ist, aber infolgedessen auch mehr Fehler im vorderen Bereich macht.

Modellqualität

Genauigkeit

Eine der einfachsten Kenngrößen für die Modellqualität logistischer Regressionsmodelle, bzw. bei der Vorhersage kategorischer oder binärer Variablen ist die Genauigkeit der Vorhersagen, also in

wie vielen Fällen, die dem Zielwert überstimmen. Der Wert wird oft relativ über der Gesamtzahl angegeben und kann einfach berechnet werden:

```
(egywth.flm_T==egywth.EV_HT_740_ON).sum() / egywth.EV_HT_740_ON.count()
```

```
np.float64(0.6056466302367942)
```

```
(egywthNoNA.flm_T==egywthNoNA.EV_HT_740_ON).sum() /
egywthNoNA.EV_HT_740_ON.count()
```

```
np.float64(0.5440758293838862)
```

Im Falle der logistischen Regression gibt die `score`-Methode die Genauigkeit (Prozentsatz der korrekten Vorhersagen) anstatt von R^2 aus.

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

```
0.5440758293838862
```

Dabei zeigt sich, dass das Statsmodell hier eine höhere Genauigkeit erreicht. Dieser Vergleich ist allerdings nur eingeschränkt aussagekräftig, da beide Modelle unterschiedliche Eingangsvariablen verwenden. Jetzt klingt die Genauigkeit von 54% Prozent erstmal „ok“. Vergleichen wir die Genauigkeit allerdings mit einer reinen Zufallsreihe. Da die Originalwerte ja recht gleich verteilt waren, können wir eine binomiale Gleichverteilung nutzen. Hier ergibt sich eine Genauigkeit von ca. 50%. Das heißt das SKlearn-Modell ist eigentlich nur in 4% der Fälle besser als eine reine Zufallswahl.

```
(np.random.randint(2,
size=egywth.EV_HT_740_ON.count())==egywth.EV_HT_740_ON).sum() /
egywth.EV_HT_740_ON.count()
```

```
np.float64(0.48360655737704916)
```

Konfusionsmatrix

Eine Konfusionsmatrix ist ein wichtiges Werkzeug zur Bewertung der Leistung von Modellen im Bereich des maschinellen Lernens zur Vorhersage von kategorischen Variablen. Sie vergleichen die vorhergesagten Werte gegen die originalen Werte.

```
confusion_matrix=egywth[["EV_HT_740_ON",
"flm_T"]].value_counts().reset_index().sort_values(['EV_HT_740_ON','flm_T'])
confusion_matrix
```

	EV_HT_740_ON	flm_T	count
0	0	0.0	460
3	0	1.0	57
1	1	0.0	333
2	1	1.0	205

```
px.imshow(confusion_matrix["count"].values.reshape(2, 2).T, x=["0","1"],
y=["0","1"], color_continuous_scale='Viridis', text_auto=True,
labels=dict(x="Ziel", y="Vorhersage", color="Anzahl"), title="Statsmodel")
```

```
Unable to display output for mime type(s): text/html
```

```
from sklearn.metrics import confusion_matrix
px.imshow(confusion_matrix(egywthNoNA.EV_HT_740_ON, egywthNoNA.flm_T).T,
x=["0","1"], y=["0","1"], color_continuous_scale='Viridis', text_auto=True,
labels=dict(x="Ziel", y="Vorhersage", color="Anzahl"), title="SKlearn")
```

```
Unable to display output for mime type(s): text/html
```

Die Einträge innerhalb der Matrix zeigen die Anzahl der Datenpunkte in jeder dieser Kategorien an. Bei einem guten Modell sind alle Vorhersagen identisch mit dem Ziel und liegen vollständig auf der Diagonalen. Werte, die von der Diagonale abweichen, sind falsche Vorhersagen. Die Konfusionmatrix erlaubt uns dabei zu verstehen, welche Vorhersagen häufig richtig bzw. falsch sind und ggf. zielgerichtet das Modell bzw. die Daten zu verbessern. Gibt es z.B. eine große Imbalance zwischen den einzelnen Kategorien, muss man ggf. den Datensatz durch Sampling neu gewichten, da sonst das Modell auf diese häufigen Kategorien übergewichtet (overfitting).

Vergleichen wir die zwei Modelle, so sehen wir, dass das Modell mit Statsmodels allgemein besser ist als das SKlearn-Modell und insbesondere die Off-Tage besser voraussagt, da es auch die Wochentage nutzt. Allerdings sagt es auch häufiger die On-Tage falsch voraus.

Wahre und falsche Vorhersagen

Die Diskussion der Konfusionsmatrix hat bereits gezeigt, dass die Genauigkeit nicht hinreichend ist, um die Modellqualität gut zu bewerten, da nicht nur wichtig ist ob etwas häufig richtig ist, sondern ob das Modell systematische Fehler macht, also zum Beispiel nur eine häufige Kategorie korrekt vorhersagt, aber bei seltenen Kategorien meist falsch liegt und somit zur Vorhersage dieser Kategorien ungeeignet ist. Deshalb unterscheidet man bei binären Problemen die Vorhersagen detailliert in:

- *True Positiv*: Korrekt als Ja klassifiziert,
- *True Negativ*: Korrekt als Nein klassifiziert,
- *False Positiv*: Falsch als Ja klassifiziert und

- *False Negativ*: Falsch als Nein klassifiziert.

```
Unable to display output for mime type(s): text/html
```

Hieraus berechnet man verschiedene Kenngrößen:

- *Genauigkeit (Accuracy)*: Die Genauigkeit ist das Verhältnis der korrekt vorhergesagten Instanzen zur Gesamtanzahl der Instanzen:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

- *Präzision (Precision)*: Die Präzision gibt an, wie viele der als positiv vorhergesagten Instanzen tatsächlich positiv sind:

$$Precision = \frac{TP}{TP + FP}$$

```
Unable to display output for mime type(s): text/html
```

- *Sensitivität (Recall) oder Trefferquote (True Positive Rate, TPR)*: Die Sensitivität misst den Anteil der tatsächlich positiven Instanzen, die korrekt als positiv vorhergesagt wurden:

$$Recall = \frac{TP}{TP + FN}$$

- *Spezifität (Specificity) oder True Negative Rate (TNR)*: Die Spezifität misst den Anteil der tatsächlich negativen Instanzen, die korrekt als negativ vorhergesagt wurden:

$$Specificity = \frac{TN}{TN + FP}$$

- *Falschalarmrate (Fallout) oder False Positive Rate (FPR)*: Die Falschalarmrate bestimmt den Anteil der falschen als Positiv vorhergesagten Werte:

$$Fallout = \frac{FP}{TN + FP}$$

```
Unable to display output for mime type(s): text/html
```

- *F1-Score*: Der F1-Score ist das harmonische Mittel von Präzision und Sensitivität und gibt eine ausgewogene Messung der Modellleistung:

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} = \frac{2TP}{2TP + FP + FN}$$

```
Unable to display output for mime type(s): text/html
```

- *Log-Loss oder Binäre Kreuzentropie*: Log-Loss misst die Leistung eines Klassifikationsmodells, dessen Vorhersagen Wahrscheinlichkeitswerte zwischen 0 und 1 sind. Es bestraft falsche Klassifikationen, insbesondere wenn die Wahrscheinlichkeiten stark danebenliegen:

$$LogLoss = -\frac{1}{n}\sum_{i=1}^{n}(y_i \log(\sigma_i) + (1 - y_i)\log(1 - \sigma_i)).$$

```
Unable to display output for mime type(s): text/html
```

Wir können die Kenngrößen mit SKLearn bestimmen:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score

print("Statsmodels")
sm = egywth[["EV_HT_740_ON", "flm_T"]].dropna()
print('Accuracy: {:.2f}'.format(accuracy_score(sm.EV_HT_740_ON, sm.flm_T)))
print('Precision: {:.2f}'.format(precision_score(sm.EV_HT_740_ON, sm.flm_T)))
print('Recall: {:.2f}'.format(recall_score(sm.EV_HT_740_ON, sm.flm_T)))
print('F1-Score: {:.2f}'.format(f1_score(sm.EV_HT_740_ON, sm.flm_T)))
```

```
Statsmodels
Accuracy: 0.63
Precision: 0.78
Recall: 0.38
F1-Score: 0.51
```

```
print("SKlearn")
print('Accuracy: {:.2f}'.format(accuracy_score(egywthNoNA.EV_HT_740_ON,
egywthNoNA.flm_T)))
print('Precision: {:.2f}'.format(precision_score(egywthNoNA.EV_HT_740_ON,
egywthNoNA.flm_T)))
print('Recall: {:.2f}'.format(recall_score(egywthNoNA.EV_HT_740_ON,
egywthNoNA.flm_T)))
print('F1-Score: {:.2f}'.format(f1_score(egywthNoNA.EV_HT_740_ON,
egywthNoNA.flm_T)))
```

```
SKlearn
Accuracy: 0.54
Precision: 0.55
Recall: 0.63
F1-Score: 0.59
```

Hier sieht man, dass das sklearn-Modell zwar eine gewisse Präzision erreicht, aber Recall und F1-Score insgesamt noch nicht überzeugend sind.

Finales Model

Um das Problem mit unserem Modell zu beheben, können wir das Jahr mit in unser Logistisches Modell integrieren.

```
egywth["Year"] = egywth["Date"].dt.year

smmY = smf.logit("EV_HT_740_ON ~ TMK + SDK + NM + VPM + Weekday + Year",
data=egywth)
smmY = smmY.fit()
```

```
Optimization terminated successfully.
         Current function value: 0.187412
         Iterations 15
```

```
smmY.summary()
```

```
egywth["flm_PY"] = smmY.predict(egywth)
egywth["flm_TY"] = egywth.flm_PY.map(lambda x: 0 if x < 0.5 else 1,
na_action='ignore')
px.imshow(egywth[["EV_HT_740_ON", "flm_TY"]].T, x=egywth["Date"],
color_continuous_scale='Viridis')
```

```
Unable to display output for mime type(s): text/html
```

Mit deutlich besserer Genauigkeit, Präzision und F1-Score.

```
print("Statsmodels Fixed")
sm = egywth[["EV_HT_740_ON", "flm_TY"]].dropna()
print('Accuracy: {:.2f}'.format(accuracy_score(sm.EV_HT_740_ON, sm.flm_TY)))
print('Precision: {:.2f}'.format(precision_score(sm.EV_HT_740_ON, sm.flm_TY)))
print('Recall: {:.2f}'.format(recall_score(sm.EV_HT_740_ON, sm.flm_TY)))
print('F1-Score: {:.2f}'.format(f1_score(sm.EV_HT_740_ON, sm.flm_TY)))
```

```
Statsmodels Fixed
Accuracy: 0.93
Precision: 0.94
Recall: 0.92
F1-Score: 0.93
```

und besserer Konfusionsmatrix.

```
px.imshow(confusion_matrix(sm.EV_HT_740_ON, sm.flm_TY).T, x=["0","1"],
y=["0","1"], color_continuous_scale='Viridis', text_auto=True,
```

```
labels=dict(x="Ziel", y="Vorhersage", color="Anzahl"), title="Statsmodels
Fixed")
```

```
Unable to display output for mime type(s): text/html
```

Bibliographie