Klassifikation

Joern Ploennigs · AI4SC

> Den Wald vor lauter Bäumen nicht sehen
>
> — Christoph Martin Wieland

Die Klassifikation ist die zweite wichtige Problemstellung im Supervised Learning deren Ziel es ist für eine Menge an Daten vorher bekannte Klassen zu lernen und diese für neue Datensätze vorherzusagen. Bei den Klassen kann es sich um binäre Daten handeln (z.B. Ja/Nein - Erkennung, ob ein Bauteil intakt oder beschädigt ist, Ein/Aus - Aktivität von Anlagen) oder kategorische Daten (z.B. Kategorisierung von Rissen: fein, mittel, grob). Sie können aber auch zur Klassifikation genutzt werden – auch wenn die mathematische Struktur Regressionsmodellen ähnelt - genauso wie logistische Regressionsmodelle, die speziell für binäre Klassifikation gedacht sind verwendet werden können.

Klassifikationsmodelle werden in der Praxis häufig zur Erkennung von Spam eingesetzt (Ja/Nein), Betrugserkennung, Anomalieerkennung, Kreditratings oder Allgemein für Vorhersageprobleme. Es gibt verschiedene Arten von Klassifikationsmodellen, von denen wir einige typische Ansätze diskutieren wollen.

Entscheidungsbäume zur Klassifikation

Entscheidungsbäume (en. *Decission Trees*) sind eine beliebte Methode des maschinellen Lernens zur Klassifizierung von Datenpunkten, da sie intuitiv und leicht verständlich sind. Das macht sie zu einem guten Werkzeug für Anfänger und Experten gleichermaßen. Sie können numerische und kategorische Daten als Eingangsvariablen verarbeiten.

Das Grundelement eines Entscheidungsbaum ist eine einfache Wenn-Dann-Sonst-Verzweigung (if-else), die wir aus der Programmierung kennen. Hierbei wird auf einer Eingangsgrößen eine Bedingung definiert und für die beiden Antwortvarianten werden unterschiedliche Rückgabewerte ausgegeben.

```
if x > 30:
  y = "Cool"
else:
  if x < 18:
    y = "Heat"
  else:
    y = "off"
```

Durch Verschachteln der Bedingungen entsteht eine Baumstruktur, die wir in einem Flussdiagramm darstellen können. Das Flussdiagramm besteht aus Knoten, Kanten und Blättern:

- **Knoten**: Ein Punkt im Baum, an dem eine Entscheidung getroffen wird, basierend auf einem Eingangswert.
- **Kanten**: Verbindungen zwischen den Knoten, die den Entscheidungsfluss darstellen.
- **Blätter**: Endknoten des Baumes, die eine Klassifikation oder Vorhersage darstellen.

Durch diesen einfachen Aufbau ist ein Entscheidungsbaum bis zu einer gewissen Tiefe gut erklärbar, da man für jeden Eingangswert die entsprechende Entscheidungskette im Baum nachvollziehen kann. Deshalb werden Entscheidungsbäume auch beim Data Mining genutzt, um bisher unbekannte Regeln aus einem Datensatz abzuleiten.

Ferner ist der Baum bei der Vorhersage (Scoring) sehr effizient zu berechnen, da durch die Baumstruktur nur die entlang des jeweiligen Pfades liegenden Bedingungen betrachtet werden müssen. Der Aufwand wächst damit im Wesentlichen mit der Tiefe des Baumes.

Erstellen eines Entscheidungsbaums

Sind die Regeln bekannt, kann der Entscheidungsbaum, wie im obigen Beispiel gezeigm, manuell definiert werden. Im Machine Learning besteht allerdings die Zielstellung diesen Entscheidungsbaum automatisch aus den Trainingsdaten abzuleiten. Das geschieht in drei Schritten:

1. Auswahl des besten Attributs zur Trennung der Daten.
2. Aufteilung des Datensatzes basierend auf dem ausgewählten Attribut.
3. Wiederholung des Prozesses für jeden Teilbaum, bis ein Abbruchkriterium erfüllt ist.

Auswahl des besten Attributes

Die Auswahl des besten Attributs erfolgt typischerweise durch Maximierung eines Kriteriums wie dem Informationsgewinn oder der Gini-Impurity.

Informationsgewinn: Der Informationsgewinn basiert auf dem Konzept der Entropie. Die Entropie misst, wie viel Unsicherheit oder Unreinheit eine Klassenverteilung enthält. Die Entropie $H(S)$ einer Variable S wird wie folgt berechnet:

$$H(S) = -\sum_{k=1}^{c} p_k \log_2(p_k)$$

wobei c die Anzahl der Klassen und p_k der Anteil der Klasse k ist, also $p_k = \frac{|C_k|}{|S|}$, wobei C_k die Menge der Instanzen der Klasse k ist.

Der Term $-\log_2(p_k)$ misst den **Informationsgehalt** in **Bit**, je seltener ein Ereignis, desto überraschender:

- $p_k = 1$: $-\log_2\left(\frac{1}{1}\right) = 0$ Bit — keine Überraschung
- $p_k = \frac{1}{2}$: $-\log_2\left(\frac{1}{2}\right) = 1$ Bit — ein Münzwurf
- $p_k = \frac{1}{4}$: $-\log_2\left(\frac{1}{4}\right) = 2$ Bit — seltener Ausgang

Die Entropie ist der **Erwartungswert** dieses Informationsgehalts über alle Klassen.

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

Der Informationsgewinn $IG(T, A)$ eines Attributs A für den Datensatz T wird dann berechnet als:

$$\overbrace{IG(T,A)}^{\text{Informationsgewinn}} = \overbrace{H(T)}^{\text{Entropy des Knoten}} - \overbrace{\sum_{v \in Values(A)} \frac{|T_v|}{|T|} H(T_v)}^{\text{Entropy aller Subknoten}}$$

wobei $Values(A)$ die möglichen Werte von Attribut A sind und T_v der Teil des Datensatzes T ist, bei dem Attribut A den Wert v hat. Der Informationsgewinn misst somit wie viel Entropie ein Knoten zu dem finalen Datensatz beiträgt.

Gini-Impurity: Die Gini-Impurity eines Datensatzes S wird wie folgt berechnet:

$$Gini(S) = 1 - \sum_{i=1}^{c} p_i^2$$

Der Gini-Gewinn wird ähnlich wie der Informationsgewinn berechnet, wobei anstelle der Entropie die Gini-Impurity verwendet wird.

Beispiel

Label:	Orange	Orange	Apfel	Orange	Orange	Apfel	Apfel	Apfel	Orange
Farbe :	orange	rot	rot	orange	orange	rot	rot	rot	orange
Größe :	groß	groß	groß	klein	klein	klein	klein	groß	klein

Betrachten wir ein einfaches Beispiel zur Veranschaulichung. Angenommen, wir haben den obigen Datensatz mit den Klassen „Apfel“ und „Orange“ und den Attributen „Farbe“ und „Größe“. Unser Ziel ist es, einen Entscheidungsbaum zu erstellen, der die Früchte klassifiziert.

Berechnung der Entropie des gesamten Datensatzes

Entsprechend der obigen Abbildung enthält der Datensatz 4 Äpfel und 5 Orangen. Die Entropie des gesamten Datensatzes ist folglich:

$$H(S) = -\left(\underbrace{\frac{5}{9}\log_2 \frac{5}{9}}_{5\text{ Orangen}} + \underbrace{\frac{4}{9}\log_2 \frac{4}{9}}_{4\text{ Äpfel}} \right) = 0.99$$

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

Berechnung der Entropie für jedes Attribut

Wir berechnen nun für jeden möglichen Split der Daten den resultierenden Informationsgewinn. Wir haben können den Datensatz anhand des Attributs `Farbe` splitten oder anhand des Attributs `Größe`. Splitten wir den Datensatz auf Basis der Farbe so haben wir 5 Objekte mit der Farbe „orange“ und 4 mit der Farbe „rot“.

Wir berechnen die Entropie nach der Aufteilung auf Basis der Farbe:

$$H(\text{Farbe=Rot}) = -\left(\underbrace{\frac{1}{5}\log_2 \frac{1}{5}}_{1\text{ Orange}} + \underbrace{\frac{4}{5}\log_2 \frac{4}{5}}_{4\text{ Äpfel}} \right) = 0.72$$

$$H(\text{Farbe=Orange}) = -\left(\underbrace{\frac{4}{4}\log_2 \frac{4}{4}}_{4\text{ Orangen}} \right) = 0.0$$

$$\begin{aligned} IG(\text{Farbe}) &= H(S) - \frac{5}{9}H(\text{Farbe=Rot}) - \frac{4}{9}H(\text{Farbe=Orange}) \\ &= 0.99 - \frac{5}{9}0.72 - \frac{4}{9}0.0 = 0.59 \end{aligned}$$

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

Jetzt berechnen wir die Entropie nach der Aufteilung auf Basis der Größe:

$$H(\text{Größe=Klein}) = -\left(\underbrace{\frac{2}{5}\log_2\frac{2}{5}}_{\text{2 Äpfel}} + \underbrace{\frac{3}{5}\log_2\frac{3}{5}}_{\text{3 Orangen}}\right) = 0.97$$

$$H(\text{Größe=Gross}) = -\left(\underbrace{\frac{2}{4}\log_2\frac{2}{4}}_{\text{2 Äpfel}} + \underbrace{\frac{2}{4}\log_2\frac{2}{4}}_{\text{2 Orangen}}\right) = 1.0$$

$$IG(\text{Größe}) = H(S) - \frac{5}{9}H(\text{Größe=Klein}) - \frac{4}{9}H(\text{Größe=Gross})$$

$$= 0.99 - \frac{5}{9}0.97 - \frac{4}{9}1.0 = 0.01$$

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

Es zeigt sich, dass der Informationsgewinn bei der $Farbe$ deutlich höher ist als bei der $Größe$, weshalb es sinnvoll ist diesen als ersten Knoten im Entscheidungsbaum zu wählen.

Entscheidungsbäume in SciKit-Learn

Entscheidungsbäume können unter andrem mit SciKit-Learn erstellt werden. Hierfür erstellen wir wie gewohnt eine Modellklasse und können dabei auch das Kriterium angeben

```
from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier(criterion="entropy")
```

Erstellen wir uns als nächstes den Beispieldatensatz as Pandas Dataframe

```
import numpy as np # Import von NumPy
import pandas as pd # Import von Pandas
df=pd.DataFrame([
    ["Orange","Gross","Orange"], ["Rot","Gross","Orange"],
["Rot","Gross","Apfel"],
    ["Orange","Klein","Orange"], ["Orange","Klein","Orange"],
["Rot","Klein","Apfel"],
    ["Rot","Klein","Apfel"], ["Rot","Gross","Apfel"],
["Orange","Klein","Orange"]
],columns=["Color","Size","Typ"])
df
```

	Color	Size	Typ
0	Orange	Gross	Orange

	Color	Size	Typ
1	Rot	Gross	Orange
2	Rot	Gross	Apfel
3	Orange	Klein	Orange
4	Orange	Klein	Orange
5	Rot	Klein	Apfel
6	Rot	Klein	Apfel
7	Rot	Gross	Apfel
8	Orange	Klein	Orange

Hier ist wieder zu beachten das SciKit-Learn nicht direkt auf kategorischen Spalten mit Text arbeitet, sondern binäre oder numerische Spalten erwartet. Wir können binäre Variablen mit der Methode `get_dummies` erzeugen:

```
dfD= pd.get_dummies(df,columns=["Color","Size","Typ"],drop_first=False)
dfD.head(2)
```

	Color_Orange	Color_Rot	Size_Gross	Size_Klein	Typ_Apfel	Typ_Orange
0	True	False	True	False	False	True
1	False	True	True	False	False	True

Oder alternativ numerische Variablen mit der Funktion `pd.factorize` erzeugen. Das bietet sich hier mehr an, da es eine kompaktere Darstellung ist

```
dfF=df.copy()
dfF["Color"], df_map_col = pd.factorize(df["Color"])
dfF["Size"], df_map_size = pd.factorize(df["Size"])
dfF["Typ"], df_map_typ = pd.factorize(df["Typ"])
dfF, df_map_col, df_map_size, df_map_typ
```

```
(   Color  Size  Typ
 0      0     0    0
 1      1     0    0
 2      1     0    1
 3      0     1    0
 4      0     1    0
 5      1     1    1
 6      1     1    1
 7      1     0    1
 8      0     1    0,
```

```
 Index(['Orange', 'Rot'], dtype='str'),
 Index(['Gross', 'Klein'], dtype='str'),
 Index(['Orange', 'Apfel'], dtype='str'))
```

Mit der Modellmethode `fit` können wir wieder das Modell trainieren, indem wir die Eingänge `x` und die Zielvariable `y` spezifizieren.

```
x=dfF[['Color','Size']]
y=dfF["Typ"]

clf.fit(x, y)
```

	criterion criterion: {"gini", "entropy", "log_loss"}, default="gini" The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "log_loss" and "entropy" both for the Shannon information gain, see :ref:`tree_mathematical_formulation`.	'entropy'
	splitter splitter: {"best", "random"}, default="best" The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.	'best'
	max_depth max_depth: int, default=None The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all	None

	leaves contain less than min_samples_split samples.	
	min_samples_split min_samples_split: int or float, default=2 The minimum number of samples required to split an internal node: - If int, then consider `min_samples_split` as the minimum number. - If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split. .. versionchanged:: 0.18 Added float values for fractions.	2
	min_samples_leaf min_samples_leaf: int or float, default=1 The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least ``min_samples_leaf`` training samples in each of the left and right branches. This may have the effect of smoothing the model,	1

	especially in regression. - If int, then consider `min_samples_leaf` as the minimum number. - If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node. .. versionchanged:: 0.18 Added float values for fractions.	
	min_weight_fraction_leaf min_weight_fraction_leaf: float, default=0.0 The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.	0.0
	max_features max_features: int, float or {"sqrt", "log2"}, default=None The number of features to consider when looking for the best split: - If int, then consider `max_features` features at each split.	None

	- If float, then `max_features` is a fraction and `max(1, int(max_features * n_features_in_))` features are considered at each split. - If "sqrt", then `max_features=sqrt(n_features)`. - If "log2", then `max_features=log2(n_features)`. - If None, then `max_features=n_features`. .. note:: The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than ``max_features`` features.	
	random_state random_state: int, RandomState instance or None, default=None Controls the randomness of the estimator. The features are always randomly permuted at each split, even if ``splitter`` is set to ``"best"``. When ``max_features < n_features``, the algorithm will select ``max_features`` at random at each split before finding the best split among them. But the best found split may vary across different runs, even if ``max_features=n_features``.	None

	That is the case, if the improvement of the criterion is identical for several splits and one split has to be selected at random. To obtain a deterministic behaviour during fitting, ``random_state`` has to be fixed to an integer. See :term:`Glossary ` for details.	
	max_leaf_nodes max_leaf_nodes: int, default=None Grow a tree with ``max_leaf_nodes`` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.	None
	min_impurity_decrease min_impurity_decrease: float, default=0.0 A node will be split if this split induces a decrease of the impurity greater than or equal to this value. The weighted impurity decrease equation is the following:: N_t / N * (impurity - N_t_R / N_t * right_impurity - N_t_L / N_t * left_impurity)	0.0

	where ``N`` is the total number of samples, ``N_t`` is the number of samples at the current node, ``N_t_L`` is the number of samples in the left child, and ``N_t_R`` is the number of samples in the right child. ``N``, ``N_t``, ``N_t_R`` and ``N_t_L`` all refer to the weighted sum, if ``sample_weight`` is passed. .. versionadded:: 0.19	
	class_weight class_weight: dict, list of dict or "balanced", default=None Weights associated with classes in the form ``{class_label: weight}``. If None, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y. Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of	None

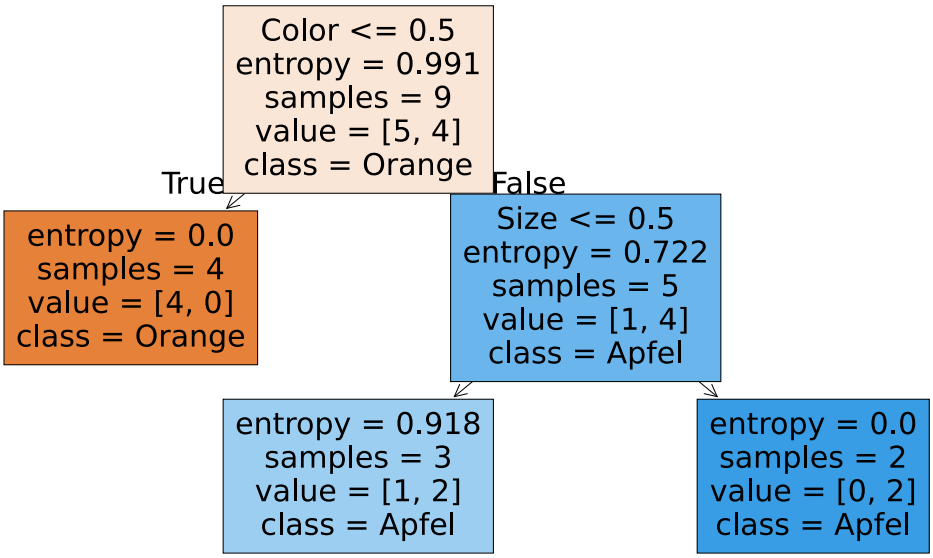
	[{1:1}, {2:5}, {3:1}, {4:1}]. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as ``n_samples / (n_classes * np.bincount(y))`` For multi-output, the weights of each column of y will be multiplied. Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.	
	ccp_alpha ccp_alpha: non-negative float, default=0.0 Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ``ccp_alpha`` will be chosen. By default, no pruning is performed. See :ref:`minimal_cost_complexity_pruning` for details. See :ref:`sphx_glr_auto_examples_tree_plot_cost_complexity_pruning.py` for an example of such pruning. .. versionadded:: 0.22	0.0
	monotonic_cst monotonic_cst: array-like of	None

	int of shape (n_features), default=None Indicates the monotonicity constraint to enforce on each feature. - 1: monotonic increase - 0: no constraint - −1: monotonic decrease If monotonic_cst is None, no constraints are applied. Monotonicity constraints are not supported for: - multiclass classifications (i.e. when `n_classes > 2`), - multioutput classifications (i.e. when `n_outputs_ > 1`), - classifications trained on data with missing values. The constraints hold over the probability of the positive class. Read more in the :ref:`User Guide `. .. versionadded:: 1.4	

Mit der Funktion `plot_tree` können wir uns den Entscheidungsbaum anzeigen lassen:

```
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

plt.figure(figsize=(20,10))
plot_tree(clf, filled=True, feature_names=['Color','Size'],
class_names=df_map_typ)
plt.show()
```



Mit der bekannten Methode `predict` können wir eine Vorhersage treffen. Also ein Objekt das Orange (0) ist und Klein (1) ist eine Orange (0).

```
clf.predict([[0,1]])
```

```
/Users/jplonnigs/Documents/code/Lehre/IntroductionMachineLearning/.venv/lib/
python3.12/site-packages/sklearn/utils/validation.py:2691: UserWarning: X does
not have valid feature names, but DecisionTreeClassifier was fitted with
feature names
  warnings.warn(
```

```
array([0])
```

Prüfen wir das Modell einmal an dem uns bekannten Energie- und Wetterdatensatz. Wir laden den Datensatz und fügen eine Spalte für den Wochentag (gleich numerisch) hinzu und ob der `EV_HT_740`-Verbraucher an ist. Wir brauchen in diesem Fall die fehlenden `NaN` Werte nicht entfernen.

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

Wir wollen ein Modell für die Temperaturklasse und den Heiztage und Kühltage bestimmen, um zu sehen ob die originalen Grenzen aus dem Notebook zur Datenvorverarbeitung erkannt werden. Hierfür wandeln wir die Spalten zuerst in Numerische Kategorien um.

```
egywth["TemperaturKlasseN"], TK_map = pd.factorize(egywth["TemperaturKlasse"])
egywth["HeizKuehlTageN"], HKT_map = pd.factorize(egywth["HeizKuehlTage"])
TK_map,HKT_map
```

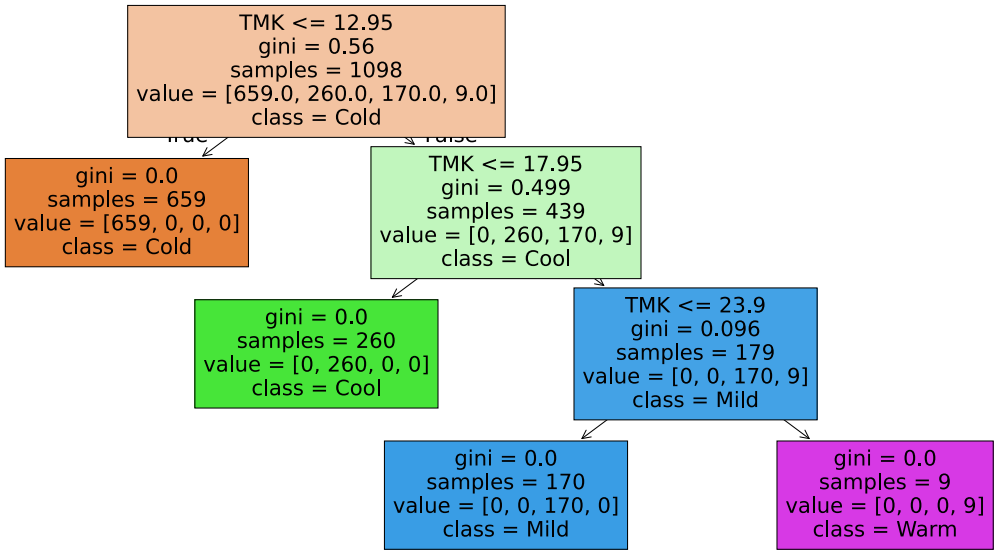
```
(Index(['Cold', 'Cool', 'Mild', 'Warm'], dtype='str'),
 Index(['Heizgradtag', 'Normaltag', 'Kühlgradtag'], dtype='str'))
```

Dann trainieren und visualisieren wir die Entscheidungsbäume.

```
clfTK = DecisionTreeClassifier()
clfTK.fit(egywth[['TMK']], egywth[['TemperaturKlasseN']])
clfTK.get_depth()
```

```
3
```

```
plt.figure(figsize=(20,10))
plot_tree(clfTK, filled=True, feature_names=['TMK'], class_names=TK_map)
plt.show()
```

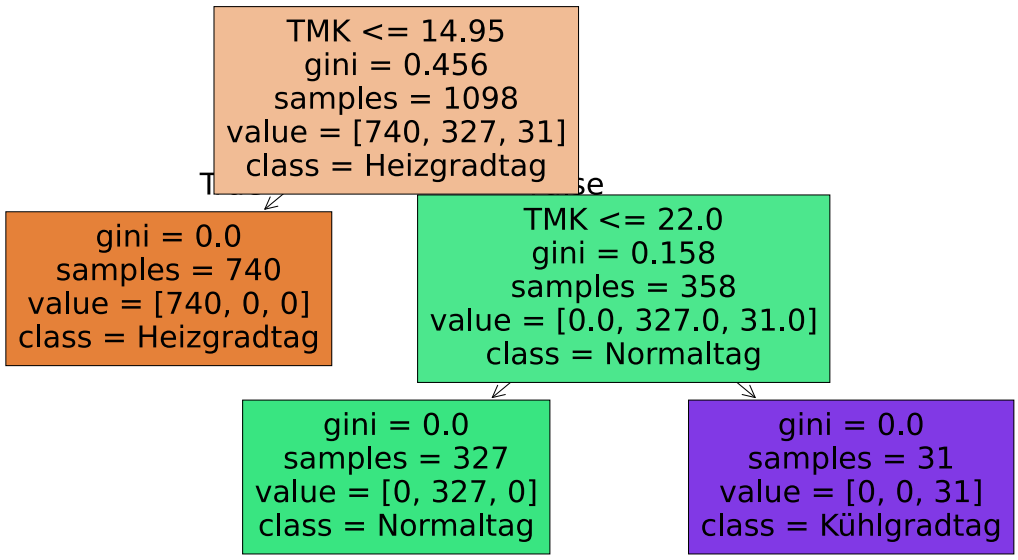


Dann trainieren und visualisieren wir die Entscheidungsbäume.

```
clfHKT = DecisionTreeClassifier()
clfHKT.fit(egywth[['TMK']], egywth[['HeizKuehlTageN']])
clfHKT.get_depth()
```

```
2
```

```
plt.figure(figsize=(20,10))
plot_tree(clfHKT, filled=True, feature_names=['TMK'], class_names=HKT_map)
plt.show()
```



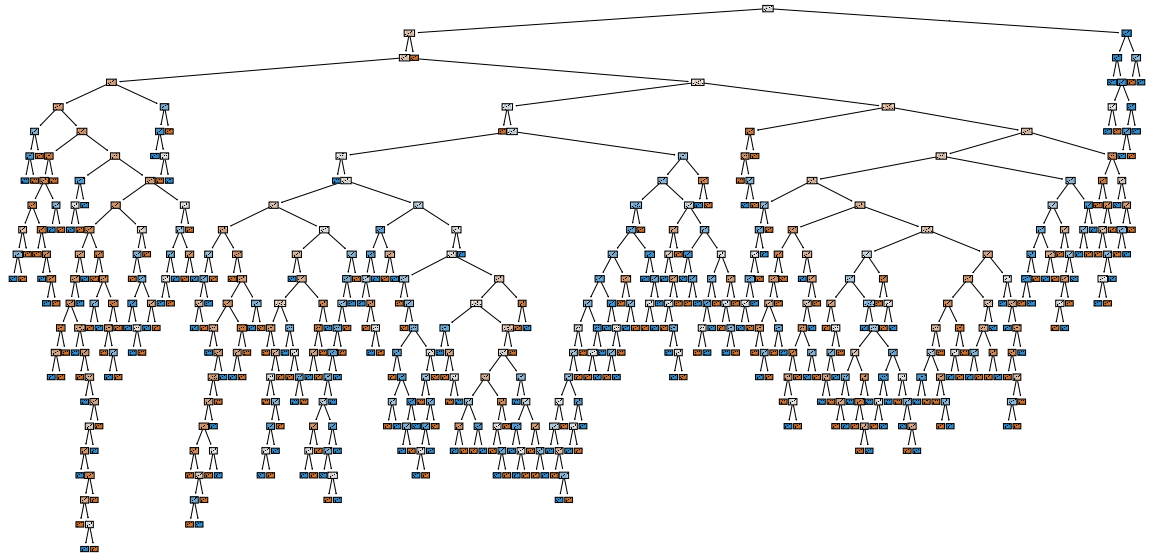
Es zeigt sich, dass die originalen Grenzen und die Folge der Werte durchaus gut identifiziert werden, wenn auch nicht ganz präzise, da entsprechende Samples im Datensatz fehlen.

Trainieren wir einen Entscheidungsbaum für die Aktivität des `EV_HT_740`-Verbrauches, wie bei der logistischen Regression.

```
clfHT = DecisionTreeClassifier()
clfHT.fit(egywth[["SDK", "NM", "VPM", "TMK", "Weekday"]],
egywth[['EV_HT_740_IS_ON']])
clfHT.get_depth()
```

```
22
```

```
plt.figure(figsize=(20,10))
plot_tree(clfHT, filled=True, feature_names=["SDK", "NM", "VPM", "TMK",
"Weekday"], class_names=['Off',"On"])
plt.show()
```



Das resultiert in einen deutlich komplexeren Entscheidungsbaum mit einer Tiefe von 21 Knoten.

Prüfen wir einmal die Vorhersagequalität, indem wir den Trainingsdatensatz vorhersagen.

```
egywth["pred_DT"] = clfHT.predict(egywth[["SDK", "NM", "VPM", "TMK",
"Weekday"]])
```

Als Vergleichsmetriken für Entscheidungsbäume die kategorische Werte vorhersagen, verwendet man die Metriken, die wir für die Logistische Regression diskutiert haben.

```
from sklearn.metrics import accuracy_score, f1_score

print('Accuracy: {:.2f}'.format(accuracy_score(egywth.EV_HT_740_IS_ON.values,
egywth.pred_DT)))
print('F1-Score: {:.2f}'.format(f1_score(egywth.EV_HT_740_IS_ON,
egywth.pred_DT)))
```

```
Accuracy: 1.00
F1-Score: 1.00
```

Hier zeigt sich, dass der Entscheidungsbaum den Ein/Aus-Zustand perfekt vorhersagen kann. Die Logistischen Regressionsmodelle im Vergleich erreichten nur eine Genauigkeit von 54% und 60%.

Es ist zu beachten, dass Modelle mit dieser perfekten Qualität meist nicht gewollt sind, da sie Überfitten (Overfitting) und dann nicht gut generalisieren. Generalisierung bedeutet, dass das Modell auch gut anwendbar auf neue Datensätze ist.

Overfitting, Train-Test-Split und Kreuz-validation

Um das Überfitting besser zu erkennen und die Generalisierbarkeit eines Modelles zu bewerten, verwendet man üblicherweise zur Bewertung und zum Vergleich von Modellen nicht einfach den Trainingsdatensatz, sondern teilt die Daten in einen Datensatz zum Trainieren und einem zum Testen (Train-Test-Split). Dabei darf der Testdatensatz keine Trainingsdaten enthalten.

Teilen wir den Datensatz einmal in einen Test- und einen Trainingsdatensatz mit jeweils 60% und 40% Daten auf und trainieren das Modell nur auf dem Trainingsdatensatz.

```
egywth_train=egywth.head(int(egywth.shape[0]*6/10)).copy()
egywth_test=egywth.tail(int(egywth.shape[0]*4/10)).copy()
```

```
clfHT_T = DecisionTreeClassifier()
clfHT_T.fit(egywth_train[["SDK", "NM", "VPM", "TMK", "Weekday"]],
egywth_train[['EV_HT_740_IS_ON']])
clfHT_T.get_depth()
```

```
13
```

```
egywth_train["pred_DT"] = clfHT_T.predict(egywth_train[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Train Accuracy:
{:.2f}'.format(accuracy_score(egywth_train.EV_HT_740_IS_ON.values,
egywth_train.pred_DT)))
```

```
Train Accuracy: 1.00
```

Wir erhalten wieder ein Modell, das zu 100% korrekt vorhersagt.

Wenden wir das Modell einmal auf dem Testdatensatz an.

```
egywth_test["pred_DT"] = clfHT_T.predict(egywth_test[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Test Accuracy:
{:.2f}'.format(accuracy_score(egywth_test.EV_HT_740_IS_ON.values,
egywth_test.pred_DT)))
```

```
Test Accuracy: 0.16
```

Wir sehen jetzt, dass das Modell auf einmal sehr schlecht ist mit einer Genauigkeit von nur 16% (wir bedenken, dass eine Zufallswahl bereits 50% Genauigkeit hat). Das Modell, das also im Training perfekt war, ist im Test komplett schlecht. Das liegt zum einen daran, dass wir in der Logistischen Regression bereits gesehen haben, dass der Energieverbrauch im hinteren Bereich

nahezu immer aktiv ist, also der Testdatensatz nicht dem Trainingsdatensatz gut entspricht. Dadurch lernt das Modell ein Verhalten, das im Testdatensatz gar nicht mehr vorkommt.

Dies können wir verhindern, indem wir die Trainings und Testdaten zufällig auswählen. Wenn die Zielklassen unausgeglichen sind, ist es zudem sinnvoll, die Aufteilung zu stratifizieren, damit Trainings- und Testdatensatz eine ähnliche Klassenverteilung behalten.

```
from sklearn.model_selection import train_test_split

egywthN=egywth.dropna()  # Wir entfernen hier wieder die NaN Werte, da einige
Ensemble-Methoden keine NaN Werte unterstützen
egywth_train, egywth_test = train_test_split(egywthN, test_size=0.4,
stratify=egywthN["EV_HT_740_IS_ON"], random_state=42)
```

```
clfHT_T = DecisionTreeClassifier()
clfHT_T.fit(egywth_train[["SDK", "NM", "VPM", "TMK", "Weekday"]],
egywth_train[['EV_HT_740_IS_ON']])
clfHT_T.get_depth()
```

```
22
```

Wenden wir das Modell zum Vergleich auf den Trainings- und Testdatensatz an.

```
egywth_train["pred_DT"] = clfHT_T.predict(egywth_train[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Train Accuracy:
{:.2f}'.format(accuracy_score(egywth_train.EV_HT_740_IS_ON.values,
egywth_train.pred_DT)))
```

```
Train Accuracy: 1.00
```

```
egywth_test["pred_DT"] = clfHT_T.predict(egywth_test[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Test Accuracy:
{:.2f}'.format(accuracy_score(egywth_test.EV_HT_740_IS_ON.values,
egywth_test.pred_DT)))
```

```
Test Accuracy: 0.60
```

Um abzusichern, dass man nicht zufällig einen besonders guten oder schlechten Trainings- und Testdatensatz auswählt, empfiehlt es sich, das Sampling und Training systematisch auf mehreren Teilmengen zu wiederholen. Dieses Verfahren nennt man k-Kreuzvalidation (k-Cross-Validation),

wobei k die Anzahl der Folds beziehungsweise Teilmengen angibt. Auch hierfür bietet `sklearn` eine vorhandene Unterstützungsfunktion an.

```
from sklearn.model_selection import cross_val_score

cross_val_score(clfHT_T, egywth_train[["SDK", "NM", "VPM", "TMK", "Weekday"]],
egywth_train[['EV_HT_740_IS_ON']], cv=5)
```

```
array([0.55454545, 0.6       , 0.56363636, 0.56363636, 0.64220183])
```

Hier sehen, wir das trotz der Zufallsauswahl das für die Trainingsdaten ideale Modell immer noch nicht gut die Testdaten vorhersagen kann, es also nicht nur daran liegt, das nicht genug Daten verfügbar sind, sondern, das overfitted Modell einfach sich relativ schlecht generalisieren lässt. Das ist eine typische Eigenschaft von Entscheidungsbäumen.

Das Problem lässt sich reduzieren, wenn man die Tiefe der Bäume reduziert.

```
clfHT_T2 = DecisionTreeClassifier(max_depth=10)
clfHT_T2.fit(egywth_train[["SDK", "NM", "VPM", "TMK", "Weekday"]],
egywth_train[['EV_HT_740_IS_ON']])
clfHT_T2.get_depth()
```

```
10
```

Wenden wir das Modell wieder zum Vergleich auf den Trainings- und Testdatensatz an.

```
egywth_train["pred_DT2"] = clfHT_T2.predict(egywth_train[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Train Accuracy:
{:.2f}'.format(accuracy_score(egywth_train.EV_HT_740_IS_ON.values,
egywth_train.pred_DT2)))
```

```
Train Accuracy: 0.87
```

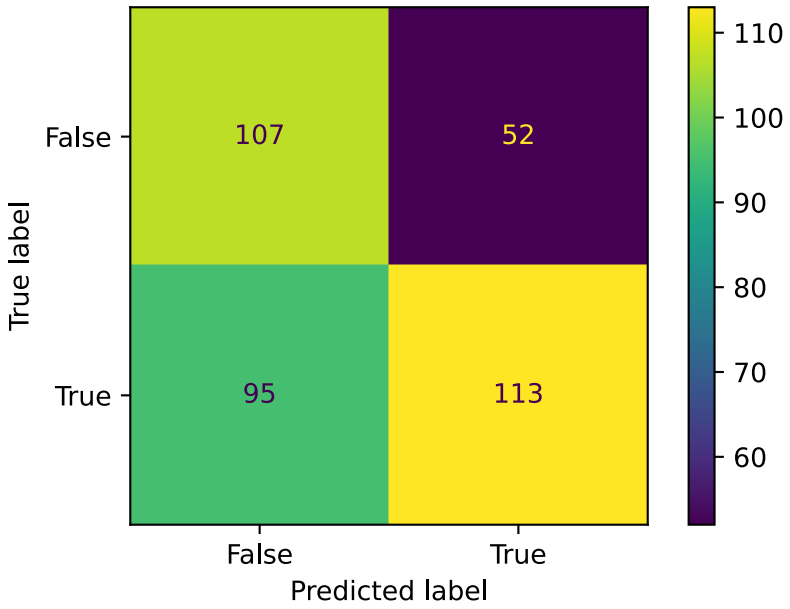
```
egywth_test["pred_DT2"] = clfHT_T2.predict(egywth_test[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Test Accuracy:
{:.2f}'.format(accuracy_score(egywth_test.EV_HT_740_IS_ON.values,
egywth_test.pred_DT2)))
```

```
Test Accuracy: 0.60
```

Zur genaueren Beurteilung betrachten wir zusätzlich eine *Confusion Matrix*. Sie zeigt, welche Klassen richtig erkannt wurden und bei welchen Klassenarten das Modell häufiger Fehler macht.

```
from sklearn.metrics import ConfusionMatrixDisplay

ConfusionMatrixDisplay.from_predictions(
    egywth_test.EV_HT_740_IS_ON,
    egywth_test.pred_DT2
)
plt.show()
```



Dadurch wird zwar die Qualität des Modells auf dem Trainingsdatensatz schlechter, aber überraschenderweise steigt meist die Qualität auf dem Testdatensatz.

Kombinationsmodelle (Ensemble Models)

Wir haben am obigen Beispiel gesehen, dass Entscheidungsbäume stark zu Overfitting neigen. Deshalb hat man verschiedene Modellvarianten entwickelt, welche die Generalisierbarkeit und somit Gesamtgenauigkeit verbessern, aber die Vorteile des einfachen Modellkonzeptes beibehalten.

Diese Modellvarianten nutzen die Idee mehrere einfache Modelle zu kombinieren. Sie werden als *Ensemble-Modells* bezeichnet. Das Hauptziel von Ensemble-Modellen ist es, die Vorhersagegenauigkeit zu erhöhen und die Schwächen einzelner Modelle zu kompensieren (wie Overfitting), da durch die Kombination mehrerer Modelle tendenziell die Varianz reduziert wird. Der Nachteil von Ensemble-Modellen ist der erhöhte Rechenbedarfs. Insbesondere beim Stacking und Voting werden vollständige Modelle trainiert, wodurch der Rechenaufwand steigt.

Es gibt verschiedene Ansätze zur Erstellung von Ensemble-Modellen, die sich in der Art und Weise unterscheiden, wie die einzelnen Modelle kombiniert werden:

- **Bagging (Bootstrap Aggregation):** Mehrere Modelle werden unabhängig voneinander auf verschiedenen Stichproben des Trainingsdatensatzes trainiert, die durch Ziehen mit Zurücklegen (Bootstrap-Sampling) erzeugt werden.
- **Boosting:** Modelle werden sequenziell trainiert, wobei jedes neue Modell darauf abzielt, die Fehler der vorherigen Modelle zu korrigieren. Jedes Modell wird somit fokussierter und trägt dazu bei, die Gesamtleistung zu verbessern.
- **Stacking:** Mehrere Modelle unterschiedlichen Typs (Basislerner) werden parallel trainiert um ihre Stärken zu kombinieren. Dieses Metamodell lernt, die Vorhersagen der Basislerner zu kombinieren, um eine endgültige Vorhersage zu treffen.
- **Voting:** Die Vorhersagen mehrerer Modelle werden kombiniert, indem über die Vorhersagen abgestimmt wird. Für Klassifikationsprobleme kann dies Mehrheitsabstimmung (die Klasse mit den meisten Stimmen wird ausgewählt) sein, während für Regressionsprobleme eine Mittelung der Vorhersagen vorgenommen werden kann.

Bagging/Bootstrap - Random Forest

Random Forests ist eine der beliebtesten Ensemble-Modelle. Es ist eine Bagging-Variante eines Entscheidungsbaumes, bei dem mehrere kleinere Bäume auf zufällige Ausschnitte des Datensatzes trainiert, dadurch soll das Overfitting vermieden werden und somit die Genauigkeit bei der Generalisierung verbessert werden. Wir können uns prinzipiell einen Random Forest selbst aus einzelnen Entscheidungsbäumen erstellen.

```
forest=[]
for i in range(50):
    egywth_train_sub=egywth_train.sample(int(0.2*egywth_train.shape[0]))
    tree = DecisionTreeClassifier(max_depth=4)
    tree.fit(egywth_train_sub[["SDK", "NM", "VPM", "TMK", "Weekday"]],
egywth_train_sub.EV_HT_740_IS_ON.values)
    forest.append(tree)
```

Wir sehen, dass die einzelnen Bäume kleiner sind, da sie einzeln weniger Fälle modellieren müssen.

Beim Bootstrap-Sampling wird jede Stichprobe durch Ziehen **mit Zurücklegen** erzeugt — jeder Baum sieht einen anderen Ausschnitt des Datensatzes:

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

Grau = nicht in Stichprobe · ×2 = doppelt gezogen · im Schnitt ~63% eindeutige Punkte pro Stichprobe

Zur Vorhersage berechnen wir das Mehrheitsvotum der Vorhersage aller Bäume bei diskreten Daten und den Mittelwert der Vorhersage der einzelnen Bäume bei numerischen Daten bei Regressionsmodellen.

```
pred=np.zeros(egywth_train.shape[0])
for tree in forest:
    pred += tree.predict(egywth_train[["SDK", "NM", "VPM", "TMK",
"Weekday"]]).astype(int)
egywth_train['pred_forest']= pred > len(forest)/2 # da dies eine binäre
vorhersage ist berechnen wir den Mehreitsvote

print('Train Accuracy:
{:.2f}'.format(accuracy_score(egywth_train.EV_HT_740_IS_ON.values,
egywth_train.pred_forest)))
```

```
Train Accuracy: 0.71
```

```
pred=np.zeros(egywth_test.shape[0])
for tree in forest:
    pred += tree.predict(egywth_test[["SDK", "NM", "VPM", "TMK", "Weekday"]])
egywth_test['pred_forest']= pred > len(forest)/2 # Mehreitsvote

print('Test Accuracy:
{:.2f}'.format(accuracy_score(egywth_test.EV_HT_740_IS_ON.values,
egywth_test.pred_forest)))
```

```
Test Accuracy: 0.61
```

In der Praxis nutzt man den fertigen Wald aus `sklearn.ensemble`, welche standartmäßig mit 100 Bäumen arbeitet.

```
from sklearn.ensemble import RandomForestClassifier
clf_RF = RandomForestClassifier()
clf_RF.fit(egywth_train[["SDK", "NM", "VPM", "TMK", "Weekday"]],
egywth_train.EV_HT_740_IS_ON.values)
```

	n_estimators n_estimators: int, default=100 The number of trees in the forest. .. versionchanged:: 0.22 The default value of ``n_estimators`` changed	100

	from 10 to 100 in 0.22.	
	criterion criterion: {"gini", "entropy", "log_loss"}, default="gini" The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "log_loss" and "entropy" both for the Shannon information gain, see :ref:`tree_mathematical_formulation`. Note: This parameter is tree-specific.	'gini'
	max_depth max_depth: int, default=None The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.	None
	min_samples_split min_samples_split: int or float, default=2 The minimum number of samples required to split an internal node: - If int, then consider `min_samples_split` as the minimum number. - If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum	2

	number of samples for each split. .. versionchanged:: 0.18 Added float values for fractions.	
	min_samples_leaf min_samples_leaf: int or float, default=1 The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least ``min_samples_leaf`` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression. - If int, then consider `min_samples_leaf` as the minimum number. - If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node. .. versionchanged:: 0.18 Added float values for fractions.	1
	min_weight_fraction_leaf min_weight_fraction_leaf:	0.0

	float, default=0.0 The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.	
	max_features max_features: {"sqrt", "log2", None}, int or float, default="sqrt" The number of features to consider when looking for the best split: - If int, then consider `max_features` features at each split. - If float, then `max_features` is a fraction and `max(1, int(max_features * n_features_in_))` features are considered at each split. - If "sqrt", then `max_features=sqrt(n_features)`. - If "log2", then `max_features=log2(n_features)`. - If None, then `max_features=n_features`. .. versionchanged:: 1.1 The default of `max_features` changed from `"auto"` to `"sqrt"`. Note: the search for a split does not stop until at least one	'sqrt'

	valid partition of the node samples is found, even if it requires to effectively inspect more than ``max_features`` features.	
	max_leaf_nodes max_leaf_nodes: int, default=None Grow trees with ``max_leaf_nodes`` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.	None
	min_impurity_decrease min_impurity_decrease: float, default=0.0 A node will be split if this split induces a decrease of the impurity greater than or equal to this value. The weighted impurity decrease equation is the following:: N_t / N * (impurity - N_t_R / N_t * right_impurity - N_t_L / N_t * left_impurity) where ``N`` is the total number of samples, ``N_t`` is the number of samples at the current node, ``N_t_L`` is the number of samples in the left child, and ``N_t_R`` is	0.0

	the number of samples in the right child. ``N``, ``N_t``, ``N_t_R`` and ``N_t_L`` all refer to the weighted sum, if ``sample_weight`` is passed. .. versionadded:: 0.19	
	bootstrap bootstrap: bool, default=True Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.	True
	oob_score oob_score: bool or callable, default=False Whether to use out-of-bag samples to estimate the generalization score. By default, :func:`~sklearn.metrics.accuracy_score` is used. Provide a callable with signature `metric(y_true, y_pred)` to use a custom metric. Only available if `bootstrap=True`. For an illustration of out-of-bag (OOB) error estimation, see the example :ref:`sphx_glr_auto_examples_ensemble_plot_ensemble_oob.py`.	False
	n_jobs n_jobs: int, default=None The number of jobs to run in parallel. :meth:`fit`, :me-	None

	th:`predict`, :meth:`decision_path` and :meth:`apply` are all parallelized over the trees. ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context. ``-1`` means using all processors. See :term:`Glossary ` for more details.	
	random_state random_state: int, RandomState instance or None, default=None Controls both the randomness of the bootstrapping of the samples used when building trees (if ``bootstrap=True``) and the sampling of the features to consider when looking for the best split at each node (if ``max_features < n_features``). See :term:`Glossary ` for details.	None
	verbose verbose: int, default=0 Controls the verbosity when fitting and predicting.	0
	warm_start warm_start: bool, default=False When set to ``True``, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole	False

	new forest. See :term:`Glossary ` and :ref:`tree_ensemble_warm_start` for details.	
	class_weight class_weight: {"balanced", "balanced_subsample"}, dict or list of dicts, default=None Weights associated with classes in the form ``{class_label: weight}``. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y. Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}]. 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))``	None

	The "balanced_subsample" mode is the same as "balanced" except that weights are computed based on the bootstrap sample for every tree grown. For multi-output, the weights of each column of y will be multiplied. Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.	
	ccp_alpha ccp_alpha: non-negative float, default=0.0 Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ``ccp_alpha`` will be chosen. By default, no pruning is performed. See :ref:`minimal_cost_complexity_pruning` for details. See :ref:`sphx_glr_auto_examples_tree_plot_cost_complexity_pruning.py` for an example of such pruning. .. versionadded:: 0.22	0.0
	max_samples max_samples: int or float, default=None If bootstrap is True, the number of samples to draw from X	None

	to train each base estimator. - If None (default), then draw `X.shape[0]` samples. - If int, then draw `max_samples` samples. - If float, then draw `max(round(n_samples * max_samples), 1)` samples. Thus, `max_samples` should be in the interval `(0.0, 1.0]`. .. versionadded:: 0.22	
	monotonic_cst monotonic_cst: array-like of int of shape (n_features), default=None Indicates the monotonicity constraint to enforce on each feature. - 1: monotonic increase - 0: no constraint - −1: monotonic decrease If monotonic_cst is None, no constraints are applied. Monotonicity constraints are not supported for: - multiclass classifications (i.e. when `n_classes > 2`), - multioutput classifications (i.e. when `n_outputs_ > 1`), - classifications trained on data with missing values. The constraints hold over the probability of the positive class.	None

	Read more in the :ref:`User Guide `. .. versionadded:: 1.4	

Mit der bekannten Predict Methode können wir die Vorhersage treffen und die Genauigkeit berechnen.

```
egywth_train["pred_RF"] = clf_RF.predict(egywth_train[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Train Accuracy:
{:.2f}'.format(accuracy_score(egywth_train.EV_HT_740_IS_ON.values,
egywth_train.pred_RF)))
```

```
Train Accuracy: 1.00
```

```
egywth_test["pred_RF"] = clf_RF.predict(egywth_test[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Test Accuracy:
{:.2f}'.format(accuracy_score(egywth_test.EV_HT_740_IS_ON.values,
egywth_test.pred_RF)))
```

```
Test Accuracy: 0.58
```

Das Ergebnis ist schon besser als der einfache Entscheidungsbaum.

Boosting

Boosting ist eine andere Methode zum Erstellen von Ensemble-Modellen, bei der eine Reihe schwacher Modelle (d. h. Modelle, die nur geringfügig besser als Zufallsvorhersagen sind) sequenziell trainiert und kombiniert werden, um ein starkes Modell zu erstellen. Die Grundidee besteht darin, aufeinanderfolgende Modelle so zu trainieren, dass sie die Fehler der vorhergehenden Modelle korrigieren.

Aber Achtung: Da Boosting-Algorithmen stark auf schwerwiegende Fehler fokussieren, können sie empfindlich gegenüber Ausreißern sein.

Jedes Modell trainiert auf dem **gesamten** Datensatz, aber mit angepassten Gewichten: falsch klassifizierte Punkte erhalten mehr Gewicht und werden im nächsten Modell stärker berücksichtigt:

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

Punktgröße ∝ Gewicht · Rote Umrandung + × = in dieser Runde falsch klassifiziert (erhält höheres Gewicht)

Adaboost

AdaBoost ist eines der bekanntesten boosting Modelle. Es beginnt mit einem schwachen Modell, das auf den gesamten Datensatz trainiert wird. In den folgenden Iterationen werden die Gewichte der falsch klassifizierten Datenpunkte erhöht und ein neues Modell wird trainiert. Dies wiederholt sich, und die Modelle werden so kombiniert, dass sie schwerwiegendere Fehler korrigieren. AdaBoost ist einfach zu implementieren und funktioniert gut mit vielen Basislernern, wie Entscheidungsstümpfen (einfachen Entscheidungsbäumen).

```
from sklearn.ensemble import AdaBoostClassifier
clf_AB = AdaBoostClassifier()
clf_AB.fit(egywth_train[["SDK", "NM", "VPM", "TMK", "Weekday"]],
egywth_train.EV_HT_740_IS_ON.values)
```

	estimator estimator: object, default=None The base estimator from which the boosted ensemble is built. Support for sample weighting is required, as well as proper ``classes_`` and ``n_classes_`` attributes. If ``None``, then the base estimator is :class:`~sklearn.tree.DecisionTreeClassifier` initialized with `max_depth=1`. .. versionadded:: 1.2 `base_estimator` was renamed to `estimator`.	None
	n_estimators n_estimators: int, default=50 The maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early. Values must be in the range `[1, inf)`.	50

	learning_rate learning_rate: float, default=1.0 Weight applied to each classifier at each boosting iteration. A higher learning rate increases the contribution of each classifier. There is a trade-off between the `learning_rate` and `n_estimators` parameters. Values must be in the range `(0.0, inf)`.	1.0
	random_state random_state: int, RandomState instance or None, default=None Controls the random seed given at each `estimator` at each boosting iteration. Thus, it is only used when `estimator` exposes a `random_state`. Pass an int for reproducible output across multiple function calls. See :term:`Glossary `.	None

Wenden wir das Modell auf den Trainings- und Testdatensatz an.

```
egywth_train["pred_AB"] = clf_AB.predict(egywth_train[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Train Accuracy:
{:.2f}'.format(accuracy_score(egywth_train.EV_HT_740_IS_ON.values,
egywth_train.pred_AB)))
```

```
Train Accuracy: 0.63
```

```
egywth_test["pred_AB"] = clf_AB.predict(egywth_test[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Test Accuracy:
{:.2f}'.format(accuracy_score(egywth_test.EV_HT_740_IS_ON.values,
egywth_test.pred_AB)))
```

```
Test Accuracy: 0.61
```

Gradient Boosting

Gradient Boosting optimiert ein beliebiges Verlustmaß (z. B. die quadratische Fehlerfunktion bei Regression oder die logarithmische Verlustfunktion bei Klassifikation) durch iteratives Hinzufügen von Modellen, die die negativen Gradienten (Richtungen des steilsten Abstiegs) der Verlustfunktion vorhersagen. Gradient Boosting ist flexibler und leistungsfähiger als AdaBoost, da es eine bessere Optimierung des Verlustmaßes ermöglicht.

```
from sklearn.ensemble import GradientBoostingClassifier
clf_GB = GradientBoostingClassifier()
clf_GB.fit(egywth_train[["SDK", "NM", "VPM", "TMK", "Weekday"]],
egywth_train.EV_HT_740_IS_ON.values)
```

	loss loss: {'log_loss', 'exponential'}, default='log_loss' The loss function to be optimized. 'log_loss' refers to binomial and multinomial deviance, the same as used in logistic regression. It is a good choice for classification with probabilistic outputs. For loss 'exponential', gradient boosting recovers the AdaBoost algorithm.	'log_loss'
	learning_rate learning_rate: float, default=0.1 Learning rate shrinks the contribution of each tree by `learning_rate`.	0.1

	There is a trade-off between learning_rate and n_estimators. Values must be in the range `[0.0, inf)`. For an example of the effects of this parameter and its interaction with ``subsample``, see :ref:`sphx_glr_auto_examples_ensemble_plot_gradient_boosting_regularization.py	
	n_estimators n_estimators: int, default=100 The number of boosting stages to perform. Gradient boosting is fairly robust to over-fitting so a large number usually results in better performance. Values must be in the range `[1, inf)`.	100
	subsample subsample: float, default=1.0 The fraction of samples to be used for fitting the individual base learners. If smaller than 1.0 this results in Stochastic Gradient Boosting. `subsample` interacts with the parameter `n_estimators`. Choosing `subsample < 1.0` leads to a reduction of variance and an increase in bias. Values must be in the range `(0.0, 1.0]`.	1.0

	criterion criterion: {'friedman_mse', 'squared_error'}, default='friedman_mse' The function to measure the quality of a split. Supported criteria are 'friedman_mse' for the mean squared error with improvement score by Friedman, 'squared_error' for mean squared error. The default value of 'friedman_mse' is generally the best as it can provide a better approximation in some cases. .. versionadded:: 0.18	'friedman_mse'
	min_samples_split min_samples_split: int or float, default=2 The minimum number of samples required to split an internal node: - If int, values must be in the range `[2, inf)`. - If float, values must be in the range `(0.0, 1.0]` and `min_samples_split` will be `ceil(min_samples_split * n_samples)`. .. versionchanged:: 0.18 Added float values for fractions.	2

	min_samples_leaf min_samples_leaf: int or float, default=1 The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least ``min_samples_leaf`` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression. - If int, values must be in the range `[1, inf)`. - If float, values must be in the range `(0.0, 1.0)` and `min_samples_leaf` will be `ceil(min_samples_leaf * n_samples)`. .. versionchanged:: 0.18 Added float values for fractions.	1
	min_weight_fraction_leaf min_weight_fraction_leaf: float, default=0.0 The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when	0.0

	sample_weight is not provided. Values must be in the range `[0.0, 0.5]`.	
	max_depth max_depth: int or None, default=3 Maximum depth of the individual regression estimators. The maximum depth limits the number of nodes in the tree. Tune this parameter for best performance; the best value depends on the interaction of the input variables. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. If int, values must be in the range `[1, inf)`.	3
	min_impurity_decrease min_impurity_decrease: float, default=0.0 A node will be split if this split induces a decrease of the impurity greater than or equal to this value. Values must be in the range `[0.0, inf)`. The weighted impurity decrease equation is the following:: N_t / N * (impurity - N_t_R /	0.0

	N_t * right_impurity - N_t_L / N_t * left_impurity) where ``N`` is the total number of samples, ``N_t`` is the number of samples at the current node, ``N_t_L`` is the number of samples in the left child, and ``N_t_R`` is the number of samples in the right child. ``N``, ``N_t``, ``N_t_R`` and ``N_t_L`` all refer to the weighted sum, if ``sample_weight`` is passed. .. versionadded:: 0.19	
	init init: estimator or 'zero', default=None An estimator object that is used to compute the initial predictions. ``init`` has to provide :term:`fit` and :term:`predict_proba`. If 'zero', the initial raw predictions are set to zero. By default, a ``DummyEstimator`` predicting the classes priors is used.	None
	random_state random_state: int, RandomState instance or None, default=None Controls the random seed given to each Tree estimator at each	None

	boosting iteration. In addition, it controls the random permutation of the features at each split (see Notes for more details). It also controls the random splitting of the training data to obtain a validation set if `n_iter_no_change` is not None. Pass an int for reproducible output across multiple function calls. See :term:`Glossary `.	
	max_features max_features: {'sqrt', 'log2'}, int or float, default=None The number of features to consider when looking for the best split: - If int, values must be in the range `[1, inf)`. - If float, values must be in the range `(0.0, 1.0]` and the features considered at each split will be `max(1, int(max_features * n_features_in_))`. - If 'sqrt', then `max_features=sqrt(n_features)`. - If 'log2', then `max_features=log2(n_features)`. - If None, then `max_features=n_features`. Choosing `max_features < n_features` leads to a reduction of variance and an increase in bias.	None

	Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than ``max_features`` features.	
	verbose verbose: int, default=0 Enable verbose output. If 1 then it prints progress and performance once in a while (the more trees the lower the frequency). If greater than 1 then it prints progress and performance for every tree. Values must be in the range `[0, inf)`.	0
	max_leaf_nodes max_leaf_nodes: int, default=None Grow trees with ``max_leaf_nodes`` in best-first fashion. Best nodes are defined as relative reduction in impurity. Values must be in the range `[2, inf)`. If `None`, then unlimited number of leaf nodes.	None
	warm_start warm_start: bool, default=False When set to ``True``, reuse the solution of the previous	False

	call to fit and add more estimators to the ensemble, otherwise, just erase the previous solution. See :term:`the Glossary `.	
	validation_fraction validation_fraction: float, default=0.1 The proportion of training data to set aside as validation set for early stopping. Values must be in the range `(0.0, 1.0)`. Only used if ``n_iter_no_change`` is set to an integer. .. versionadded:: 0.20	0.1
	n_iter_no_change n_iter_no_change: int, default=None ``n_iter_no_change`` is used to decide if early stopping will be used to terminate training when validation score is not improving. By default it is set to None to disable early stopping. If set to a number, it will set aside ``validation_fraction`` size of the training data as validation and terminate training when validation score is not improving in all of the previous ``n_iter_no_change`` numbers of	None

	iterations. The split is stratified. Values must be in the range `[1, inf)`. See :ref:`sphx_glr_auto_examples_ensemble_plot_gradient_boosting_early_stopping.p .. versionadded:: 0.20	
	tol tol: float, default=1e-4 Tolerance for the early stopping. When the loss is not improving by at least tol for ``n_iter_no_change`` iterations (if set to a number), the training stops. Values must be in the range `[0.0, inf)`. .. versionadded:: 0.20	0.0001
	ccp_alpha ccp_alpha: non-negative float, default=0.0 Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ``ccp_alpha`` will be chosen. By default, no pruning is performed. Values must be in the range `[0.0, inf)`. See :ref:`minimal_cost_complexity_pruning` for details. See :ref:`sphx_glr_auto_examples_tree_plot_cost_complexity_pruning.py` for an example of such pruning.	0.0

	.. versionadded:: 0.22	

Wir vergleichen das Modell auf den Trainings- und Testdatensatz an.

```
egywth_train["pred_GB"] = clf_GB.predict(egywth_train[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Train Accuracy:
{:.2f}'.format(accuracy_score(egywth_train.EV_HT_740_IS_ON.values,
egywth_train.pred_GB)))
```

```
Train Accuracy: 0.87
```

```
egywth_test["pred_GB"] = clf_GB.predict(egywth_test[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Test Accuracy:
{:.2f}'.format(accuracy_score(egywth_test.EV_HT_740_IS_ON.values,
egywth_test.pred_GB)))
```

```
Test Accuracy: 0.62
```

XGBoost

XGBoost ist eine erweiterte Version des Gradient Boosting, die auf Effizienz und Geschwindigkeit optimiert ist. Es verwendet Techniken wie Sparsity Awareness, paralleles Tree-Boosting und reguläre Begriffe, um Overfitting zu vermeiden. XGBoost ist sehr beliebt bei großen Datensätzen aufgrund seiner hohen Leistung und Effizienz.

XGBoost ist direkt nicht in `sklearn` verfügbar. Das `xgboost` bietet aber die gleiche API an.

```
import xgboost as xgb

clf_XGB = xgb.XGBClassifier()
clf_XGB.fit(egywth_train[["SDK", "NM", "VPM", "TMK", "Weekday"]],
egywth_train.EV_HT_740_IS_ON.values)
```

	objective objective: typing.Union[str, xgboost.sklearn._SklObjWProto, typing.Callable[[typing.Any, typing.Any], typing.Tuple[numpy.ndarray, numpy.ndarray]], NoneType]	'binary:logistic'

	Specify the learning task and the corresponding learning objective or a custom objective function to be used. For custom objective, see :doc:`/tutorials/custom_metric_obj` and :ref:`custom-obj-metric` for more information, along with the end note for function signatures.	
	base_score base_score: typing.Union[float, typing.List[float], NoneType] The initial prediction score of all instances, global bias.	None
	booster	None
	callbacks callbacks: typing.Optional[typing.List[xgboost.callback.TrainingCallback] List of callback functions that are applied at end of each iteration. It is possible to use predefined callbacks by using :ref:`Callback API `. .. note:: States in callback are not preserved during training, which means callback objects can not be reused for multiple training sessions without reinitialization or deepcopy. .. code-block:: python for params in	None

	parameters_grid:
	# be sure to (re)initialize the callbacks before each run
	callbacks = [xgb.callback.LearningRateScheduler(custom_rates)]
	reg = xgboost.XGBRegressor(**params, callbacks=callbacks)
	reg.fit(X, y)	

	colsample_bylevel colsample_bylevel: typing.Optional[float] Subsample ratio of columns for each level.	None
	colsample_bynode colsample_bynode: typing.Optional[float] Subsample ratio of columns for each split.	None
	colsample_bytree colsample_bytree: typing.Optional[float] Subsample ratio of columns when constructing each tree.	None
	device device: typing.Optional[str] .. versionadded:: 2.0.0 Device ordinal, available options are `cpu`, `cuda`, and `gpu`.	None
	early_stopping_rounds early_stopping_rounds: typing.Optional[int] .. versionadded:: 1.6.0	None

	- Activates early stopping. Validation metric needs to improve at least once in every **early_stopping_rounds** round(s) to continue training. Requires at least one item in **eval_set** in :py:meth:`fit`. - If early stopping occurs, the model will have two additional attributes: :py:attr:`best_score` and :py:attr:`best_iteration`. These are used by the :py:meth:`predict` and :py:meth:`apply` methods to determine the optimal number of trees during inference. If users want to access the full model (including trees built after early stopping), they can specify the `iteration_range` in these inference methods. In addition, other utilities like model plotting can also use the entire model. - If you prefer to discard the trees after `best_iteration`, consider using the callback function :py:class:`xgboost.callback.EarlyStopping`. - If there's more than one item in **eval_set**, the last entry will be used for early stopping. If there's more than one metric in	

	eval_metric, the last metric will be used for early stopping.	
	enable_categorical enable_categorical: bool See the same parameter of :py:class:`DMatrix` for details.	False
	eval_metric eval_metric: typing.Union[str, typing.List[typing.Union[str, typing.Callable]], typing.Callable, NoneType] .. versionadded:: 1.6.0 Metric used for monitoring the training result and early stopping. It can be a string or list of strings as names of predefined metric in XGBoost (See :doc:`/parameter`), one of the metrics in :py:mod:`sklearn.metrics`, or any other user defined metric that looks like `sklearn.metrics`. If custom objective is also provided, then custom metric should implement the corresponding reverse link function. Unlike the `scoring` parameter commonly used in scikit-learn, when a callable object is provided, it's assumed to be a cost function and	None

	by default XGBoost will minimize the result during early stopping. For advanced usage on Early stopping like directly choosing to maximize instead of minimize, see :py:obj:`xgboost.callback.EarlyStopping`. See :doc:`/tutorials/custom_metric_obj` and :ref:`custom-obj-metric` for more information. .. code-block:: python from sklearn.datasets import load_diabetes from sklearn.metrics import mean_absolute_error X, y = load_diabetes(return_X_y=True) reg = xgb.XGBRegressor(tree_method="hist", eval_metric=mean_absolute_error,) reg.fit(X, y, eval_set=[(X, y)])	
	feature_types feature_types: typing.Optional[typing.Sequence[str]] .. versionadded:: 1.7.0 Used for specifying feature types without constructing a dataframe. See the :py:class:`DMatrix` for details.	None
	feature_weights feature_weights: Optional[ArrayLike] Weight for each feature, defi-	None

	nes the probability of each feature being selected when colsample is being used. All values must be greater than 0, otherwise a `ValueError` is thrown.	
	gamma gamma: typing.Optional[float] (min_split_loss) Minimum loss reduction required to make a further partition on a leaf node of the tree.	None
	grow_policy grow_policy: typing.Optional[str] Tree growing policy. - depthwise: Favors splitting at nodes closest to the node, - lossguide: Favors splitting at nodes with highest loss change.	None
	importance_type	None
	interaction_constraints interaction_constraints: typing.Union[str, typing.List[typing.Tuple[str]], NoneType] Constraints for interaction representing permitted interactions. The constraints must be specified in the form of a nested list, e.g. ``[[0, 1], [2, 3, 4]]``, where each inner list is a group of indices of features that are allowed to interact with each	None

	other. See :doc:`tutorial ` for more information	
	learning_rate learning_rate: typing.Optional[float] Boosting learning rate (xgb's "eta")	None
	max_bin max_bin: typing.Optional[int] If using histogram-based algorithm, maximum number of bins per feature	None
	max_cat_threshold max_cat_threshold: typing.Optional[int] .. versionadded:: 1.7.0 .. note:: This parameter is experimental Maximum number of categories considered for each split. Used only by partition-based splits for preventing over-fitting. Also, `enable_categorical` needs to be set to have categorical feature support. See :doc:`Categorical Data ` and :ref:`cat-param` for details.	None
	max_cat_to_onehot max_cat_to_onehot: Optional[int] .. versionadded:: 1.6.0 .. note:: This parameter is ex-	None

	perimental A threshold for deciding whether XGBoost should use one-hot encoding based split for categorical data. When number of categories is lesser than the threshold then one-hot encoding is chosen, otherwise the categories will be partitioned into children nodes. Also, `enable_categorical` needs to be set to have categorical feature support. See :doc:`Categorical Data ` and :ref:`cat-param` for details.	
	max_delta_step max_delta_step: typing.Optional[float] Maximum delta step we allow each tree's weight estimation to be.	None
	max_depth max_depth: typing.Optional[int] Maximum tree depth for base learners.	None
	max_leaves max_leaves: typing.Optional[int] Maximum number of leaves; 0 indicates no limit.	None
	min_child_weight min_child_weight: typing.Optional[float] Minimum sum of instance	None

	weight(hessian) needed in a child.	
	missing missing: float Value in the data which needs to be present as a missing value. Default to :py:data:`numpy.nan`.	nan
	monotone_constraints monotone_constraints: typing.Union[typing.Dict[str, int], str, NoneType] Constraint of variable monotonicity. See :doc:`tutorial ` for more information.	None
	multi_strategy multi_strategy: typing.Optional[str] .. versionadded:: 2.0.0 .. note:: This parameter is working-in-progress. The strategy used for training multi-target models, including multi-target regression and multi-class classification. See :doc:`/tutorials/multioutput` for more information. - ``one_output_per_tree``: One model for each target. - ``multi_output_tree``: Use multi-target trees.	None
	n_estimators n_estimators: Optional[int]	None

	Number of boosting rounds.	
	n_jobs n_jobs: typing.Optional[int] Number of parallel threads used to run xgboost. When used with other Scikit-Learn algorithms like grid search, you may choose which algorithm to parallelize and balance the threads. Creating thread contention will significantly slow down both algorithms.	None
	num_parallel_tree	None
	random_state random_state: typing.Union[numpy.random.mtrand.RandomState, numpy.random._generator.Generator, int, NoneType] Random number seed. .. note:: Using gblinear booster with shotgun updater is nondeterministic as it uses Hogwild algorithm.	None
	reg_alpha reg_alpha: typing.Optional[float] L1 regularization term on weights (xgb's alpha).	None
	reg_lambda reg_lambda: typing.Optional[float] L2 regularization term on weights (xgb's lambda).	None

	sampling_method sampling_method: typing.Optional[str] Sampling method. Used only by the GPU version of ``hist`` tree method. - ``uniform``: Select random training instances uniformly. - ``gradient_based``: Select random training instances with higher probability when the gradient and hessian are larger. (cf. CatBoost)	None
	scale_pos_weight scale_pos_weight: typing.Optional[float] Balancing of positive and negative weights.	None
	subsample subsample: typing.Optional[float] Subsample ratio of the training instance.	None
	tree_method tree_method: typing.Optional[str] Specify which tree method to use. Default to auto. If this parameter is set to default, XGBoost will choose the most conservative option available. It's recommended to study this option from the parameters document :doc:`tree method `	None

	validate_parameters validate_parameters: typing.Optional[bool] Give warnings for unknown parameter.	None
	verbosity verbosity: typing.Optional[int] The degree of verbosity. Valid values are 0 (silent) - 3 (debug).	None

Vergleichen wir das Modell auf den Trainings- und Testdatensatz an.

```
egywth_train["pred_XGB"] = clf_XGB.predict(egywth_train[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Train Accuracy:
{:.2f}'.format(accuracy_score(egywth_train.EV_HT_740_IS_ON.values,
egywth_train.pred_XGB)))
```

```
Train Accuracy: 1.00
```

```
egywth_test["pred_XGB"] = clf_XGB.predict(egywth_test[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Accuracy:
{:.2f}'.format(accuracy_score(egywth_test.EV_HT_740_IS_ON.values,
egywth_test.pred_XGB)))
```

```
Accuracy: 0.61
```

Stacking

Beim Stacking kombiniert man einfach mehrere Modelle durch ein weiteres einfaches Modell. Zum Beispiel wollen wir die Vorhersage des Logistischen Regressionsmodells mit dem Decission Tree kombinieren innerhalb eines weiteren Logistischen Modells kombinieren. Hier können wir insbesondere gut ausnutzen, dass `sklearn` immer die gleiche API anbietet, wir können also das Training und die Vorhersage einfach als Loop implementieren.

```
from sklearn.linear_model import LogisticRegression

# 1. Basismodel
```

```
clfLR = LogisticRegression()
# 2. Basismodel
clfDT = DecisionTreeClassifier()
# Stacking
stack=[clfLR, clfDT]
```

Das Meta-Modell lernt, **welchem Basismodell es wann vertrauen soll** — je nach Muster der Basisvorhersagen:

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

Grüner Rand = beide Modelle korrekt · Blau/Orange = nur ein Modell richtig, Meta-Modell wählt das bessere · Roter Rand = beide falsch (nicht korrigierbar)

Wir trainineren alle Basismodelle

```
for model in stack:
    model.fit(egywth_train[["SDK", "NM", "VPM", "TMK", "Weekday"]],
egywth_train.EV_HT_740_IS_ON.values)
```

Die Modelle kombinieren wir nun in einem Stackingmodell, was meist ein sehr einfaches logistisches oder lineares Modell ist. Hierfür berechnen wir erst für den Trainingsdatensatz die Vorhersagen und geben die in ein neues gestacktes Modell, das nur die Vorhersagen enthält. Dadurch gewichten wir einfach die Basismodelle in ihrer Güte.

```
pred_train={}
for i,model in enumerate(stack):
    pred_train[f"pred{i}"]=model.predict(egywth_train[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
pred_train=pd.DataFrame(pred_train)
```

```
# 4. Stacking Model
clfST = LogisticRegression()
clfST.fit(pred_train, egywth_train.EV_HT_740_IS_ON.values)
```

	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	'deprecated'

	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 penalty 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	1.0

	effect of tuning the `C` parameter with an L1 penalty, see: :ref:`sphx_glr_auto_examples_linear_model_plot_logistic_path.py`.	
	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-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.	0.0
	dual dual: bool, default=False Dual (constrained) or primal	False

	(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.	
	tol tol: float, default=1e-4 Tolerance for stopping criteria.	0.0001
	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 fea-	1

	tures. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) `intercept_scaling` has to be increased.	
	class_weight class_weight: dict or 'balanced', default=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'*	None
	random_state random_state: int, RandomState instance, default=None Used when ``solver`` == 'sag', 'saga' or 'liblinear' to shuffle the	None

	data. See :term:`Glossary ` for details.	
	solver solver: {'lbfgs', 'liblinear', 'newton-cg', 'newton-cholesky', 'sag', 'saga'}, default='lbfgs' Algorithm to use in the optimization problem. Default is '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	'lbfgs'

	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 `. .. versionadded:: 0.17 *warm_start* to support *lbfgs*, *newton-cg*, *sag*, *saga* solvers.	False
	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

Bei der Vorhersage machen wir das gleiche und berechnen erst die Vorhersage der Basismodell und kombinieren dann diese in einer Vorhersage durch das Stackingmodell.

```
pred_test={}
for i,model in enumerate(stack):
    pred_test[f"pred{i}"]=model.predict(egywth_test[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
pred_test=pd.DataFrame(pred_test)
egywth_test['pred_stack_lm']= clfST.predict(pred_test)

print('Test Accuracy:
{:.2f}'.format(accuracy_score(egywth_test.EV_HT_740_IS_ON.values,
egywth_test.pred_stack_lm)))
```

```
Test Accuracy: 0.60
```

Auch hierfür bietet `sklearn` eine einfache Funktion an. Anstatt eines Mehrheitsvotums kombinieren wir die einzelnen Modelle hier allerdings mit einem Logistischen Modell, wodurch die Modelle in ihrer Qualität gewichtet werden und das Gesamtergebnis besser wird.

```
from sklearn.ensemble import StackingClassifier

clf_STK = StackingClassifier(estimators=[('lr', LogisticRegression()),
('dt',DecisionTreeClassifier())], final_estimator=LogisticRegression())
clf_STK.fit(egywth_train[["SDK", "NM", "VPM", "TMK", "Weekday"]],
egywth_train.EV_HT_740_IS_ON.values)
```

	estimators estimators: list of (str, estimator) Base estimators which will be stacked together. Each element of the list is defined as a tuple of string (i.e. name) and an estimator instance. An estimator can be set to 'drop' using `set_params`. The type of estimator is generally expected to be a classifier. However, one can pass a regressor for some use case (e.g. ordinal regression).	[('lr', ...), ('dt', ...)]
	final_estimator final_estimator: estimator, default=None A classifier which will be used to combine the base estimators. The default classifier is a :class:`~sklearn.linear_model.LogisticRegression`.	LogisticRegression()
	cv cv: int, cross-validation generator, iterable, or "prefit",	None

	default=None Determines the cross-validation splitting strategy used in `cross_val_predict` to train `final_estimator`. Possible inputs for cv are: * None, to use the default 5-fold cross validation, * integer, to specify the number of folds in a (Stratified) KFold, * An object to be used as a cross-validation generator, * An iterable yielding train, test splits, * `"prefit"`, to assume the `estimators` are prefit. In this case, the estimators will not be refitted. For integer/None inputs, if the estimator is a classifier and y is either binary or multiclass, :class:`~sklearn.model_selection.StratifiedKFold` is used. In all other cases, :class:`~sklearn.model_selection.KFold` is used. These splitters are instantiated with `shuffle=False` so the splits will be the same across calls. Refer :ref:`User Guide ` for the various cross-validation strategies that can be used here. If "prefit" is passed, it is assu-	

	med that all `estimators` have been fitted already. The `final_estimator_` is trained on the `estimators` predictions on the full training set and are **not** cross validated predictions. Please note that if the models have been trained on the same data to train the stacking model, there is a very high risk of overfitting. .. versionadded:: 1.1 The 'prefit' option was added in 1.1 .. note:: A larger number of split will provide no benefits if the number of training samples is large enough. Indeed, the training time will increase. ``cv`` is not used for model evaluation but for prediction.	
	stack_method stack_method: {'auto', 'predict_proba', 'decision_function', 'predict'}, default='auto' Methods called for each base estimator. It can be: * if 'auto', it will try to invoke, for each estimator, `'predict_proba'`, `'decision_function'` or `'pre-	'auto'

	dict'` in that order. * otherwise, one of `'predict_proba'`, `'decision_function'` or `'predict'`. If the method is not implemented by the estimator, it will raise an error.	
	n_jobs n_jobs: int, default=None The number of jobs to run in parallel for `fit` of all `estimators`. `None` means 1 unless in a `joblib.parallel_backend` context. -1 means using all processors. See :term:`Glossary ` for more details.	None
	passthrough passthrough: bool, default=False When False, only the predictions of estimators will be used as training data for `final_estimator`. When True, the `final_estimator` is trained on the predictions as well as the original training data.	False
	verbose verbose: int, default=0 Verbosity level.	0

	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 penalty 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'`.	'deprecated'
	C C: float, default=1.0 Inverse of regularization strength; must be a positive	1.0

	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`.	
	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-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	0.0

	`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 Tolerance for stopping criteria.	0.0001
	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 appen-	1

	ded 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.	
	class_weight class_weight: dict or 'balanced', default=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.	None

	.. 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'. 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	'lbfgs'

	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 iterati-	100

	ons taken for the solvers to converge.	
	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 `. .. versionadded:: 0.17 *warm_start* to support *lbfgs*, *newton-cg*, *sag*, *saga* solvers.	False
	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

	criterion criterion: {"gini", "entropy", "log_loss"}, default="gini" The function to measure the quality of a split. Supported	'gini'

	criteria are "gini" for the Gini impurity and "log_loss" and "entropy" both for the Shannon information gain, see :ref:`tree_mathematical_formulation`.	
	splitter splitter: {"best", "random"}, default="best" The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.	'best'
	max_depth max_depth: int, default=None The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.	None
	min_samples_split min_samples_split: int or float, default=2 The minimum number of samples required to split an internal node: - If int, then consider `min_samples_split` as the minimum number. - If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum	2

	number of samples for each split. .. versionchanged:: 0.18 Added float values for fractions.	
	min_samples_leaf min_samples_leaf: int or float, default=1 The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least ``min_samples_leaf`` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression. - If int, then consider `min_samples_leaf` as the minimum number. - If float, then `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node. .. versionchanged:: 0.18 Added float values for fractions.	1
	min_weight_fraction_leaf min_weight_fraction_leaf:	0.0

	float, default=0.0 The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.	
	max_features max_features: int, float or {"sqrt", "log2"}, default=None The number of features to consider when looking for the best split: - If int, then consider `max_features` features at each split. - If float, then `max_features` is a fraction and `max(1, int(max_features * n_features_in_))` features are considered at each split. - If "sqrt", then `max_features=sqrt(n_features)`. - If "log2", then `max_features=log2(n_features)`. - If None, then `max_features=n_features`. .. note:: The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to	None

	effectively inspect more than ``max_features`` features.	
	random_state random_state: int, RandomState instance or None, default=None Controls the randomness of the estimator. The features are always randomly permuted at each split, even if ``splitter`` is set to ``"best"``. When ``max_features < n_features``, the algorithm will select ``max_features`` at random at each split before finding the best split among them. But the best found split may vary across different runs, even if ``max_features=n_features``. That is the case, if the improvement of the criterion is identical for several splits and one split has to be selected at random. To obtain a deterministic behaviour during fitting, ``random_state`` has to be fixed to an integer. See :term:`Glossary ` for details.	None
	max_leaf_nodes max_leaf_nodes: int, default=None Grow a tree with ``max_leaf_nodes`` in best-	None

	first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.	
	min_impurity_decrease min_impurity_decrease: float, default=0.0 A node will be split if this split induces a decrease of the impurity greater than or equal to this value. The weighted impurity decrease equation is the following:: N_t / N * (impurity - N_t_R / N_t * right_impurity - N_t_L / N_t * left_impurity) where ``N`` is the total number of samples, ``N_t`` is the number of samples at the current node, ``N_t_L`` is the number of samples in the left child, and ``N_t_R`` is the number of samples in the right child. ``N``, ``N_t``, ``N_t_R`` and ``N_t_L`` all refer to the weighted sum, if ``sample_weight`` is passed. .. versionadded:: 0.19	0.0

	class_weight class_weight: dict, list of dict or "balanced", default=None Weights associated with classes in the form ``{class_label: weight}``. If None, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y. Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}]. 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))`` For multi-output, the weights of each column of y will be multiplied. Note that these weights will be multiplied with sample_weight (passed	None

	through the fit method) if sample_weight is specified.	
	ccp_alpha ccp_alpha: non-negative float, default=0.0 Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ``ccp_alpha`` will be chosen. By default, no pruning is performed. See :ref:`minimal_cost_complexity_pruning` for details. See :ref:`sphx_glr_auto_examples_tree_plot_cost_complexity_pruning.py` for an example of such pruning. .. versionadded:: 0.22	0.0
	monotonic_cst monotonic_cst: array-like of int of shape (n_features), default=None Indicates the monotonicity constraint to enforce on each feature. - 1: monotonic increase - 0: no constraint - −1: monotonic decrease If monotonic_cst is None, no constraints are applied. Monotonicity constraints are not supported for: - multiclass classifications (i.e. when `n_classes > 2`), - multioutput classifications	None

	(i.e. when `n_outputs_ > 1`), - classifications trained on data with missing values. The constraints hold over the probability of the positive class. Read more in the :ref:`User Guide `. .. versionadded:: 1.4	

	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 penalty and solver. .. versionadded:: 0.19 l1 penalty with SAGA solver (allowing 'multinomial' + L1) .. deprecated:: 1.8 `penalty` was deprecated in	'deprecated'

	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-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::	0.0

	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 Tolerance for stopping criteria.	0.0001
	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 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	None

	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'. 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` pro-	'lbfgs'

	blems (`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 `. .. versionadded:: 0.17 *warm_start* to support *lbfgs*, *newton-cg*, *sag*, *saga* solvers.	False
	n_jobs n_jobs: int, default=None Does not have any effect.	None

	.. deprecated:: 1.8 `n_jobs` is deprecated in version 1.8 and will be removed in 1.10.	

Vergleichen wir das Modell auf den Trainings- und Testdatensatz an.

```
egywth_train["pred_STK"] = clf_STK.predict(egywth_train[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Train Accuracy:
{:.2f}'.format(accuracy_score(egywth_train.EV_HT_740_IS_ON.values,
egywth_train.pred_STK)))
```

```
Train Accuracy: 0.91
```

```
egywth_test["pred_STK"] = clf_STK.predict(egywth_test[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Test Accuracy:
{:.2f}'.format(accuracy_score(egywth_test.EV_HT_740_IS_ON.values,
egywth_test.pred_STK)))
```

```
Test Accuracy: 0.62
```

Voting

Beim Voting kombinieren wir mehrere Basismodelle durch Mehrheitsabstimmung oder Median/Mittelwertbestimmung.

Wir können zum Beispiel die Basismodelle aus dem Stacking durch Mehrheitsabstimmung kombinieren.

```
pred=np.zeros(egywth_test.shape[0])
for model in stack:
    pred += model.predict(egywth_test[["SDK", "NM", "VPM", "TMK", "Weekday"]])
egywth_test['pred_stack_vote']= pred > len(forest)/2 # Mehreitsvote

print('Test Accuracy:
{:.2f}'.format(accuracy_score(egywth_test.EV_HT_740_IS_ON.values,
egywth_test.pred_stack_vote)))
```

```
Test Accuracy: 0.43
```

Jedes Modell macht **andere** Fehler — der Mehrheitsentscheid hebt sie gegenseitig auf:

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

Rote Umrandung + × = Fehler · Punkte 0 und 15 werden von ≥2 Modellen falsch klassifiziert und bleiben auch im Votum falsch

Auch hierfür bietet `sklearn` eine Methode an.

```
from sklearn.ensemble import VotingClassifier
clf_VK = VotingClassifier(estimators=[('lr', LogisticRegression()),
('dt',DecisionTreeClassifier())], voting='hard')
clf_VK.fit(egywth_train[["SDK", "NM", "VPM", "TMK", "Weekday"]],
egywth_train.EV_HT_740_IS_ON.values)
```

	estimators estimators: list of (str, estimator) tuples Invoking the ``fit`` method on the ``VotingClassifier`` will fit clones of those original estimators that will be stored in the class attribute ``self.estimators_``. An estimator can be set to ``'drop'`` using :meth:`set_params`. .. versionchanged:: 0.21 ``'drop'`` is accepted. Using None was deprecated in 0.22 and support was removed in 0.24.	[('lr', ...), ('dt', ...)]
	voting voting: {'hard', 'soft'}, default='hard' If 'hard', uses predicted class labels for majority rule voting. Else if 'soft', predicts the class label based on the argmax of the sums of the predicted probabilities, which is recom-	'hard'

	mended for an ensemble of well-calibrated classifiers.	
	weights weights: array-like of shape (n_classifiers,), default=None Sequence of weights (`float` or `int`) to weight the occurrences of predicted class labels (`hard` voting) or class probabilities before averaging (`soft` voting). Uses uniform weights if `None`.	None
	n_jobs n_jobs: int, default=None The number of jobs to run in parallel for ``fit``. ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context. ``-1`` means using all processors. See :term:`Glossary ` for more details. .. versionadded:: 0.18	None
	flatten_transform flatten_transform: bool, default=True Affects shape of transform output only when voting='soft' If voting='soft' and flatten_transform=True, transform method returns matrix with shape (n_samples, n_classifiers * n_classes). If flatten_transform=False, it	True

	returns (n_classifiers, n_samples, n_classes).	
	verbose verbose: bool, default=False If True, the time elapsed while fitting will be printed as it is completed. .. versionadded:: 0.23	False

	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 penalty and solver. .. versionadded:: 0.19 l1 penalty with SAGA solver (allowing 'multinomial' + L1) .. deprecated:: 1.8 `penalty` was deprecated in	'deprecated'

	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-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::	0.0

	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 Tolerance for stopping criteria.	0.0001
	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 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	None

	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'. 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` pro-	'lbfgs'

	blems (`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 `. .. versionadded:: 0.17 *warm_start* to support *lbfgs*, *newton-cg*, *sag*, *saga* solvers.	False
	n_jobs n_jobs: int, default=None Does not have any effect.	None

	.. deprecated:: 1.8 `n_jobs` is deprecated in version 1.8 and will be removed in 1.10.	

	criterion criterion: {"gini", "entropy", "log_loss"}, default="gini" The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "log_loss" and "entropy" both for the Shannon information gain, see :ref:`tree_mathematical_formulation`.	'gini'
	splitter splitter: {"best", "random"}, default="best" The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.	'best'
	max_depth max_depth: int, default=None The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.	None
	min_samples_split min_samples_split: int or float, default=2	2

	The minimum number of samples required to split an internal node: - If int, then consider `min_samples_split` as the minimum number. - If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split. .. versionchanged:: 0.18 Added float values for fractions.	
	min_samples_leaf min_samples_leaf: int or float, default=1 The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least ``min_samples_leaf`` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression. - If int, then consider `min_samples_leaf` as the minimum number. - If float, then	1

	`min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node. .. versionchanged:: 0.18 Added float values for fractions.	
	min_weight_fraction_leaf min_weight_fraction_leaf: float, default=0.0 The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.	0.0
	max_features max_features: int, float or {"sqrt", "log2"}, default=None The number of features to consider when looking for the best split: - If int, then consider `max_features` features at each split. - If float, then `max_features` is a fraction and `max(1, int(max_features * n_features_in_))` features are considered at each split.	None

	- If "sqrt", then `max_features=sqrt(n_features)`. - If "log2", then `max_features=log2(n_features)`. - If None, then `max_features=n_features`. .. note:: The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than ``max_features`` features.	
	random_state random_state: int, RandomState instance or None, default=None Controls the randomness of the estimator. The features are always randomly permuted at each split, even if ``splitter`` is set to ``"best"``. When ``max_features < n_features``, the algorithm will select ``max_features`` at random at each split before finding the best split among them. But the best found split may vary across different runs, even if ``max_features=n_features``. That is the case, if the improvement of the criterion is identical for several splits and one split has to be selected at random. To obtain a determi-	None

	nistic behaviour during fitting, ``random_state`` has to be fixed to an integer. See :term:`Glossary ` for details.	
	max_leaf_nodes max_leaf_nodes: int, default=None Grow a tree with ``max_leaf_nodes`` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.	None
	min_impurity_decrease min_impurity_decrease: float, default=0.0 A node will be split if this split induces a decrease of the impurity greater than or equal to this value. The weighted impurity decrease equation is the following:: N_t / N * (impurity - N_t_R / N_t * right_impurity - N_t_L / N_t * left_impurity) where ``N`` is the total number of samples, ``N_t`` is the number of samples at the current node, ``N_t_L`` is the number of samples in the	0.0

	left child, and ``N_t_R`` is the number of samples in the right child. ``N``, ``N_t``, ``N_t_R`` and ``N_t_L`` all refer to the weighted sum, if ``sample_weight`` is passed. .. versionadded:: 0.19	
	class_weight class_weight: dict, list of dict or "balanced", default=None Weights associated with classes in the form ``{class_label: weight}``. If None, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y. Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}]. The "balanced" mode uses the values of y to automatically adjust weights inversely proportio-	None

	nal to class frequencies in the input data as ``n_samples / (n_classes * np.bincount(y))`` For multi-output, the weights of each column of y will be multiplied. Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.	
	ccp_alpha ccp_alpha: non-negative float, default=0.0 Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ``ccp_alpha`` will be chosen. By default, no pruning is performed. See :ref:`minimal_cost_complexity_pruning` for details. See :ref:`sphx_glr_auto_examples_tree_plot_cost_complexity_pruning.py` for an example of such pruning. .. versionadded:: 0.22	0.0
	monotonic_cst monotonic_cst: array-like of int of shape (n_features), default=None Indicates the monotonicity constraint to enforce on each feature.	None

	- 1: monotonic increase - 0: no constraint - −1: monotonic decrease If monotonic_cst is None, no constraints are applied. Monotonicity constraints are not supported for: - multiclass classifications (i.e. when `n_classes > 2`), - multioutput classifications (i.e. when `n_outputs_ > 1`), - classifications trained on data with missing values. The constraints hold over the probability of the positive class. Read more in the :ref:`User Guide `. .. versionadded:: 1.4	

Wie performt das Modell auf den Trainings- und Testdatensatz:

```
egywth_train["pred_VK"] = clf_VK.predict(egywth_train[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Train Accuracy:
{:.2f}'.format(accuracy_score(egywth_train.EV_HT_740_IS_ON.values,
egywth_train.pred_VK)))
```

```
Train Accuracy: 0.86
```

```
egywth_test["pred_VK"] = clf_VK.predict(egywth_test[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Test Accuracy:
{:.2f}'.format(accuracy_score(egywth_test.EV_HT_740_IS_ON.values,
egywth_test.pred_VK)))
```

```
Test Accuracy: 0.61
```

Support Vector Machines

Support Vector Machines (SVMs) sind ein weiteres Modell, das zur Klassifikation und Regression verwendet werden kann. Sie sind besonders gut geeignet für Aufgaben, bei denen die Datenpunkte durch eine nichtlineare Entscheidungsgrenze getrennt werden können (Bei linearen Grenzen kann man logistische Modelle nutzen, bei scharfen Grenzen Entscheidungsbäume). SVM zielt darauf ab, die beste Trennlinie (oder Hyperebene in höheren Dimensionen) zu finden, die die Datenpunkte verschiedener Klassen mit maximalem Abstand trennen.

Lineare SVM

Für ein einfaches binäres Klassifizierungsproblem mit zwei Klassen $y_i \in \{-1, 1\}$ und Datenpunkten $\bar{x}_i \in \mathbb{R}^n$, suchen wir eine Hyperebene, die die Daten trennt. Bei linearen SVM wir diese Hyperebene durch ein lineares Modell beschrieben:

$$b + \bar{w} \cdot \bar{x} = 0$$

Dabei ist $\bar{w}$ der Normalenvektor zur Hyperebene und b ist der Bias (Verschiebungsterm). Damit sind lineare Kernel dem logistischen Regressionsmodell sehr ähnlich und zeigt eine ähnliche lineare Entscheidungsgrenze.

Der Hyperebene teilt den Raum so, dass das Vorzeichen

- positiv ist mit $b + \bar{w} \cdot \bar{x}_i \geq 0$ für $y_i = 1$
- negativ ist mit $b + \bar{w} \cdot \bar{x}_i < 0$ für $y_i = -1$

Der Abstand zwischen den beiden nächsten Punkten beider Klassen (die als **Support Vectors** bezeichnet werden) und der Hyperebene wird als **Margin** bezeichnet. Die SVM maximiert diese Margin, was gleichbedeutend mit der Minimierung von $|\bar{w}|$ ist. Die Optimierungsaufgabe lautet daher:

$$\min_{\bar{w},b} \frac{1}{2} \parallel \bar{w} \parallel^2$$

unter den Nebenbedingungen

$$y_i(b + \bar{w} \cdot \bar{x}_i) \geq 1, \forall i.$$

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

Der Kerneltrick bei nicht-lineare SVM

Für nicht-lineare Probleme können wir die Datenpunkte in einen höherdimensionalen Raum transformieren, wo sie linear trennbar sind, so dass wir dort weiterhin ein Lineares Model nutzen können. Diese Transformation wird durch den sogenannten *Kerneltrick* erreicht. Ein Kernel ist eine Funktion $K(\bar{x}_i, \bar{x}_j)$, die das Skalarprodukt der Datenpunkte im höherdimensionalen Raum

berechnet, ohne dass die Transformation explizit durchgeführt werden muss. Beliebte Kernelfunktionen sind:

- Linearkern: $K(\bar{x}_i, \bar{x}_j) = \bar{x}_i \cdot \bar{x}_j$
- Polynomkern: $K(\bar{x}_i, \bar{x}_j) = (\bar{x}_i \cdot \bar{x}_j + 1)^d$
- RBF-Kern (Radial Basis Function): $K(\bar{x}_i, \bar{x}_j) = \exp\left(-\gamma \parallel \bar{x}_i - \bar{x}_j \parallel^2\right)$

Damit ist der Kernel vom Prinzip vergleichbar mit der Transferfunktion in einem GAM-Model, mit dem Unterschied, dass der Kernel das Skalarprodukt transformiert.

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

Um den Kernelfunktionen Rechnung zu tragen, ändert sich das Optimierungsproblem zu:

$$\min_{\bar{w},b} \frac{1}{2} \parallel \bar{w} \parallel^2 + C \sum_{i=1}^{n} \xi_i$$

unter den Nebenbedingungen

$$y_i(b + \bar{w} \cdot \phi(\bar{x}_i)) \geq 1 - \xi_i, \xi_i \geq 0, \forall i$$

Dabei ist $\phi(\bar{x})$ die Abbildung in den höheren dimensionsunabhängigen Raum und ξ_i sind die Schlupfvariablen. Diese sind notwendig, um bei nicht vollständig linear trennbaren Daten, die Fehlklassifikationen zu modellieren (gleich dem Fehlerterm ε).

Das Optimierungsproblems wird als *Duales Problem* gelöst. Die duale Formulierung ermöglicht es, die Berechnungen nur in Form von Skalarprodukten (oder Kernelfunktionen) durchzuführen:

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j K(\bar{x}_i, \bar{x}_j)$$

mit den Lagrange-Multiplikatoren α_i und unter den Nebenbedingungen

$$0 \leq \alpha_i \leq C, \sum_{i=1}^{n} \alpha_i y_i = 0.$$

Die Entscheidungsfunktion im Dualraum wird durch die Lagrange-Multiplikatoren und Kernelfunktion ausgedrückt:

$$\bar{y} = \text{sign}\left(b + \sum_{i=1}^{n} \alpha_i y_i K(\bar{x}_i, \bar{x})\right).$$

SVM in SciKit

Wir betrachten als illustratives Beispiel einmal ein zufällig erzeugtes Yin-Yang-Muster. Dies ist ein 2D-Datensatz, bei dem verschiedene Punkte unterschiedlich eingefärbt sind, und die Aufgabe besteht darin, die richtige Farbe basierend auf dem Punkt zu vorhersagen. Es handelt sich also

um eine grundlegendes Klassifikationsproblem. Um die Aufgabe jedoch hinreichend komplex zu gestalten, führen wir die Farben in einem Spiralmuster ein. Wir erstellen die Daten wie folgt:

```
import plotly.express as px

np.random.seed(33)
N = 1000  # Anzahl der Punkte
X1 = np.random.normal(size=N)
X2 = np.random.normal(size=N)
X = np.column_stack((X1, X2))
y = X1 + X2 - 1.7*np.sin(1.2*(X1 - X2)) + np.random.normal(scale=0.5, size=N)
> 0

fig=px.scatter(x=X1, y=X2, color=y, width=600, height=600,
color_discrete_sequence=["black", "white"])
fig.update_traces(marker_line=dict(width=.3, color="black"))
fig.show()
```

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

Das Bild zeigt weiße und schwarze Punkte, die ein unvollkommenes Yin-Yang-Muster mit einer unscharfen Grenze zwischen den Klassen bilden.

Der SVM-Klassifikator `SVC` ist im Paket `sklearn.svm` implementiert. Der Konstruktor der Klasse akzeptiert eine Anzahl von Argumenten, von denen die wichtigsten `kernel` sind, um verschiedene Kernel auszuwählen, sowie die entsprechenden Parameter für verschiedene Kernel, z.B. `degree` für den Polynomgrad und `gamma` für den radialen Skalierungsparameter.

Die Erstellung, das Fitting und Scoring des Modells folgen dem bekannten Muster

```
from sklearn.svm import SVC

m = SVC(kernel="linear")
_ = m.fit(X, y)
m.score(X, y)  # on training data
```

```
0.82
```

Wir erstellen uns eine Visualisierungsfunktion, um die Grenzen besser zu erkennen.

```
def DBPlot(m, X, y, nGrid = 100):
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max, nGrid),
                           np.linspace(x2_min, x2_max, nGrid))
    XX = np.column_stack((xx1.ravel(), xx2.ravel()))
```

```
    hatyy = m.predict(XX).reshape(xx1.shape)
    fig = px.imshow(hatyy, width=600, height=600)
    fig.add_scatter(x=X[:,0]/(x1_max-x1_min)*100+50, y=X[:,1]/(x2_max-
x2_min)*100+50, mode="markers",
        marker=dict(color='white', line=dict(width=.3, color="black")))
    fig.add_scatter(x=X[y,0]/(x1_max-x1_min)*100+50, y=X[y,1]/(x2_max-
x2_min)*100+50, mode="markers",
        marker=dict(color='black'))
    fig.update_coloraxes(showscale=False)
    fig.update_layout(showlegend=False)
    fig.show()
```

```
m = SVC(kernel="linear")  # linear kernel does not have important parameters
_ = m.fit(X, y)
DBPlot(m, X, y)
```

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

Das Ergebnis ist nicht zu schlecht - es erfasst den groben Unterschied der weißen und schwarzen Punkte, aber kann nicht den Kopf des Yin und Yangs modellieren.

Als nächstes wollen wir dies mit einem Polynomkern zweiten Grades probieren.

```
m = SVC(kernel="poly", degree=2)
_ = m.fit(X, y)
DBPlot(m, X, y)
m.score(X, y)
```

```
0.516
```

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

Wie Sie sehen können, ist der Polynomkern (2) in der Lage, ein blaues Band auf einem gelben Hintergrund darzustellen. Es ist fraglich, ob dies besser ist als das, was der lineare Kern leisten kann, aber man kann leicht erkennen, dass ein solches Band eine gute Darstellung für andere Arten von Daten wäre.

Als nächstes, repliziere das oben Genannte mit einem Grad-3-Kernel:

```
m = SVC(kernel="poly", degree=3)
_ = m.fit(X, y)
DBPlot(m, X, y)
m.score(X, y)
```

```
0.727
```

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

Und schließlich mit einem radialen Kernel:

```
m = SVC(kernel="rbf", gamma=1)
_ = m.fit(X, y)
DBPlot(m, X, y)
m.score(X, y)
```

```
0.907
```

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

Vergleichen wir diese mit den Entscheidungsbäumen und Ensamble-Modellen so sehen wir, dass diese den Trainingsdatensatz inklusiver der Ausnahmen gut erlernen, dadurch allerdings keine stetigen Grenzen erkennen.

```
m = DecisionTreeClassifier()
_ = m.fit(X, y)
DBPlot(m, X, y)
m.score(X, y)
```

```
1.0
```

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

```
m = RandomForestClassifier()
_ = m.fit(X, y)
DBPlot(m, X, y)
m.score(X, y)
```

```
1.0
```

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

Als nächstes testen wir die SVM-Modelle einmal auf der Energievorhersage.

```
clf_SVCL = SVC(kernel="linear")
clf_SVCL.fit(egywth_train[["SDK", "NM", "VPM", "TMK", "Weekday"]],
egywth_train.EV_HT_740_IS_ON.values)
```

	C C: float, default=1.0 Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty. For an intuitive visualization of the effects of scaling the regularization parameter C, see :ref:`sphx_glr_auto_examples_svm_plot_svm_scale_c.py`.	1.0
	kernel kernel: {'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'} or callable, default='rbf' Specifies the kernel type to be used in the algorithm. If none is given, 'rbf' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape ``(n_samples, n_samples)``. For an intuitive visualization of different kernel types see :ref:`sphx_glr_auto_examples_svm_plot_svm_kernels.py`.	'linear'
	degree degree: int, default=3 Degree of the polynomial kernel function ('poly').	3

	Must be non-negative. Ignored by all other kernels.	
	gamma gamma: {'scale', 'auto'} or float, default='scale' Kernel coefficient for 'rbf', 'poly' and 'sigmoid'. - if ``gamma='scale'`` (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma, - if 'auto', uses 1 / n_features - if float, must be non-negative. .. versionchanged:: 0.22 The default value of ``gamma`` changed from 'auto' to 'scale'.	'scale'
	coef0 coef0: float, default=0.0 Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.	0.0
	shrinking shrinking: bool, default=True Whether to use the shrinking heuristic. See the :ref:`User Guide `.	True
	probability probability: bool, default=False Whether to enable probability estimates. This must be enabled prior to calling `fit`, will slow down that method as it internally	False

	uses 5-fold cross-validation, and `predict_proba` may be inconsistent with `predict`. Read more in the :ref:`User Guide `.	
	tol tol: float, default=1e-3 Tolerance for stopping criterion.	0.001
	cache_size cache_size: float, default=200 Specify the size of the kernel cache (in MB).	200
	class_weight class_weight: dict or 'balanced', default=None Set the parameter C of class i to class_weight[i]*C for SVC. 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))``.	None
	verbose verbose: bool, default=False Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may	False

	not work properly in a multithreaded context.	
	max_iter max_iter: int, default=-1 Hard limit on iterations within solver, or −1 for no limit.	−1
	decision_function_shape decision_function_shape: {'ovo', 'ovr'}, default='ovr' Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2). However, note that internally, one-vs-one ('ovo') is always used as a multi-class strategy to train models; an ovr matrix is only constructed from the ovo matrix. The parameter is ignored for binary classification. .. versionchanged:: 0.19 decision_function_shape is 'ovr' by default. .. versionadded:: 0.17 *decision_function_shape='ovr'* is recommended.	'ovr'

	.. versionchanged:: 0.17 Deprecated *decision_function_shape='ovo' and None*.	
	break_ties break_ties: bool, default=False If true, ``decision_function_shape='ovr'``, and number of classes > 2, :term:`predict` will break ties according to the confidence values of :term:`decision_function`; otherwise the first class among the tied classes is returned. Please note that breaking ties comes at a relatively high computational cost compared to a simple predict. See :ref:`sphx_glr_auto_examples_svm_plot_svm_tie_breaking.py` for an example of its usage with ``decision_function_shape='ovr'``. .. versionadded:: 0.22	False
	random_state random_state: int, RandomState instance or None, default=None Controls the pseudo random number generation for shuffling the data for probability estimates. Ignored when `probability` is False. Pass an int for reproducible output across multiple function calls. See :term:`Glossary `.	None

Wir vergleichen das Modell auf den Trainings- und Testdatensatz an.

```
egywth_train["pred_SVCL"] = clf_SVCL.predict(egywth_train[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Train Accuracy:
{:.2f}'.format(accuracy_score(egywth_train.EV_HT_740_IS_ON.values,
egywth_train.pred_SVCL)))
```

```
Train Accuracy: 0.57
```

```
egywth_test["pred_SVCL"] = clf_SVCL.predict(egywth_test[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Test Accuracy:
{:.2f}'.format(accuracy_score(egywth_test.EV_HT_740_IS_ON.values,
egywth_test.pred_SVCL)))
```

```
Test Accuracy: 0.57
```

```
clf_SVCP = SVC(kernel="poly", degree=3)
clf_SVCP.fit(egywth_train[["SDK", "NM", "VPM", "TMK", "Weekday"]],
egywth_train.EV_HT_740_IS_ON.values)
```

	C C: float, default=1.0 Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty. For an intuitive visualization of the effects of scaling the regularization parameter C, see :ref:`sphx_glr_auto_examples_svm_plot_svm_scale_c.py`.	1.0
	kernel kernel: {'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'} or callable, default='rbf' Specifies the kernel type to be used in the algorithm. If	'poly'

	none is given, 'rbf' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape ``(n_samples, n_samples)``. For an intuitive visualization of different kernel types see :ref:`sphx_glr_auto_examples_svm_plot_svm_kernels.py`.	
	degree degree: int, default=3 Degree of the polynomial kernel function ('poly'). Must be non-negative. Ignored by all other kernels.	3
	gamma gamma: {'scale', 'auto'} or float, default='scale' Kernel coefficient for 'rbf', 'poly' and 'sigmoid'. - if ``gamma='scale'`` (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma, - if 'auto', uses 1 / n_features - if float, must be non-negative. .. versionchanged:: 0.22 The default value of ``gamma`` changed from 'auto' to 'scale'.	'scale'
	coef0 coef0: float, default=0.0 Independent term in kernel function.	0.0

	It is only significant in 'poly' and 'sigmoid'.	
	shrinking shrinking: bool, default=True Whether to use the shrinking heuristic. See the :ref:`User Guide `.	True
	probability probability: bool, default=False Whether to enable probability estimates. This must be enabled prior to calling `fit`, will slow down that method as it internally uses 5-fold cross-validation, and `predict_proba` may be inconsistent with `predict`. Read more in the :ref:`User Guide `.	False
	tol tol: float, default=1e-3 Tolerance for stopping criterion.	0.001
	cache_size cache_size: float, default=200 Specify the size of the kernel cache (in MB).	200
	class_weight class_weight: dict or 'balanced', default=None Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one.	None

	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))``.	
	verbose verbose: bool, default=False Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.	False
	max_iter max_iter: int, default=-1 Hard limit on iterations within solver, or −1 for no limit.	−1
	decision_function_shape decision_function_shape: {'ovo', 'ovr'}, default='ovr' Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2). However, note that internally, one-vs-one ('ovo')	'ovr'

	is always used as a multi-class strategy to train models; an ovr matrix is only constructed from the ovo matrix. The parameter is ignored for binary classification. .. versionchanged:: 0.19 decision_function_shape is 'ovr' by default. .. versionadded:: 0.17 *decision_function_shape='ovr'* is recommended. .. versionchanged:: 0.17 Deprecated *decision_function_shape='ovo' and None*.	
	break_ties break_ties: bool, False default=False If true, ``decision_function_shape='ovr'``, and number of classes > 2, :term:`predict` will break ties according to the confidence values of :term:`decision_function`; otherwise the first class among the tied classes is returned. Please note that breaking ties comes at a relatively high computational cost compared to a simple predict. See :ref:`sphx_glr_auto_examples_svm_plot_svm_tie_breaking.py` for an example of its usage with ``decision_function_shape='ovr'``. .. versionadded:: 0.22	

	random_state random_state: int, RandomState instance or None, default=None Controls the pseudo random number generation for shuffling the data for probability estimates. Ignored when `probability` is False. Pass an int for reproducible output across multiple function calls. See :term:`Glossary `.	None

Wir vergleichen das Modell auf den Trainings- und Testdatensatz an.

```
egywth_train["pred_SVCP"] = clf_SVCP.predict(egywth_train[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Train Accuracy:
{:.2f}'.format(accuracy_score(egywth_train.EV_HT_740_IS_ON.values,
egywth_train.pred_SVCP)))
```

```
Train Accuracy: 0.58
```

```
egywth_test["pred_SVCP"] = clf_SVCP.predict(egywth_test[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Test Accuracy:
{:.2f}'.format(accuracy_score(egywth_test.EV_HT_740_IS_ON.values,
egywth_test.pred_SVCP)))
```

```
Test Accuracy: 0.57
```

```
clf_SVCR = SVC(kernel="rbf", gamma=1)
clf_SVCR.fit(egywth_train[["SDK", "NM", "VPM", "TMK", "Weekday"]],
egywth_train.EV_HT_740_IS_ON.values)
```

	C C: float, default=1.0 Regularization parameter. The strength of the regula-	1.0

	rization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty. For an intuitive visualization of the effects of scaling the regularization parameter C, see :ref:`sphx_glr_auto_examples_svm_plot_svm_scale_c.py`.	
	kernel kernel: {'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'} or callable, default='rbf' Specifies the kernel type to be used in the algorithm. If none is given, 'rbf' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape ``(n_samples, n_samples)``. For an intuitive visualization of different kernel types see :ref:`sphx_glr_auto_examples_svm_plot_svm_kernels.py`.	'rbf'
	degree degree: int, default=3 Degree of the polynomial kernel function ('poly'). Must be non-negative. Ignored by all other kernels.	3
	gamma gamma: {'scale', 'auto'} or float, default='scale' Kernel coefficient for 'rbf', 'poly' and 'sigmoid'. - if ``gamma='scale'`` (de-	1

	fault) is passed then it uses 1 / (n_features * X.var()) as value of gamma, - if 'auto', uses 1 / n_features - if float, must be non-negative. .. versionchanged:: 0.22 The default value of ``gamma`` changed from 'auto' to 'scale'.	
	coef0 coef0: float, default=0.0 Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.	0.0
	shrinking shrinking: bool, default=True Whether to use the shrinking heuristic. See the :ref:`User Guide `.	True
	probability probability: bool, default=False Whether to enable probability estimates. This must be enabled prior to calling `fit`, will slow down that method as it internally uses 5-fold cross-validation, and `predict_proba` may be inconsistent with `predict`. Read more in the :ref:`User Guide `.	False
	tol tol: float, default=1e-3	0.001

	Tolerance for stopping criterion.	
	cache_size cache_size: float, default=200 Specify the size of the kernel cache (in MB).	200
	class_weight class_weight: dict or 'balanced', default=None Set the parameter C of class i to class_weight[i]*C for SVC. 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))``.	None
	verbose verbose: bool, default=False Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.	False
	max_iter max_iter: int, default=-1 Hard limit on iterations within solver, or −1 for no limit.	−1

	decision_function_shape decision_function_shape: {'ovo', 'ovr'}, default='ovr' Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2). However, note that internally, one-vs-one ('ovo') is always used as a multi-class strategy to train models; an ovr matrix is only constructed from the ovo matrix. The parameter is ignored for binary classification. .. versionchanged:: 0.19 decision_function_shape is 'ovr' by default. .. versionadded:: 0.17 *decision_function_shape='ovr'* is recommended. .. versionchanged:: 0.17 Deprecated *decision_function_shape='ovo' and None*.	'ovr'
	break_ties break_ties: bool, default=False If true, ``decision_function_shape='ovr'``, and number of classes > 2,	False

	:term:`predict` will break ties according to the confidence values of :term:`decision_function`; otherwise the first class among the tied classes is returned. Please note that breaking ties comes at a relatively high computational cost compared to a simple predict. See :ref:`sphx_glr_auto_examples_svm_plot_svm_tie_breaking.py` for an example of its usage with ``decision_function_shape='ovr'``. .. versionadded:: 0.22	
	random_state random_state: int, RandomState instance or None, default=None Controls the pseudo random number generation for shuffling the data for probability estimates. Ignored when `probability` is False. Pass an int for reproducible output across multiple function calls. See :term:`Glossary `.	None

Wir vergleichen das Modell auf den Trainings- und Testdatensatz an.

```
egywth_train["pred_SVCR"] = clf_SVCR.predict(egywth_train[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Train Accuracy:
{:.2f}'.format(accuracy_score(egywth_train.EV_HT_740_IS_ON.values,
egywth_train.pred_SVCR)))
```

```
Train Accuracy: 0.98
```

```
egywth_test["pred_SVCR"] = clf_SVCR.predict(egywth_test[["SDK", "NM", "VPM",
"TMK", "Weekday"]])
print('Test Accuracy:
{:.2f}'.format(accuracy_score(egywth_test.EV_HT_740_IS_ON.values,
egywth_test.pred_SVCR)))
```

```
Test Accuracy: 0.54
```

Die Vorhersagequalität dieser Modelle ist nicht sehr hoch. Das liegt daran, dass Trennlinien im Datensatz nicht gegeben sind. In solchen Fällen sind Entscheidungsbäume und Ensambles besser geeignet, obwohl sie wie gezeigt schnell zum Overfitting neigen. Dennoch gehören SVM mit zu den beliebtesten Modellen.

Deshalb empfiehlt es sich immer vorher ein Scatter-Diagramm mit der Zielklasse als Farbcode zu erstellen, um zu sehen, ob sich Grenzen erkennen lassen.

```
fig=px.scatter_matrix(egywth, dimensions=["SDK", "NM", "VPM", "TMK"],
opacity=.2, color="EV_HT_740_IS_ON")
fig.update_layout(legend=dict(orientation="h",yanchor="top",y=-0.1,xanchor="center",x=0.5))
fig.show()
```

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

Regressionsbäume und Regressions-Ensemble-Modelle

Wir können Entscheidungsbäume und Ensemble-Modelle auch als Regressionsmodell benutzen. Hierbei soll nicht eine spezifische Klasse einer kategorischen Variable vorhergesagt werden, sondern eine numerische Variable. Das Prinzip der Regressionsvarianten ist das gleiche wie beiden diskreten Modellen, nur werden die Ergebnisse nicht durch Mehrheitsabstimmung kombiniert, sondern linear addiert.

Vergleichen wir die oben genannten Varianten einmal in ihren Modell-Varianten und sagen den Energieverbrauch `ES_Lab` voraus.

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor,
StackingRegressor, VotingRegressor
from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, root_mean_squared_error,
r2_score
import time

reg_models=[("Linear Model", LinearRegression()),
            ("Decision Tree", DecisionTreeRegressor()),
```

```
            ("Random Forest", RandomForestRegressor()),
            ("Gradient Boost", GradientBoostingRegressor()),
            ("XGBoost", xgb.XGBRegressor()),
            ("Stack", StackingRegressor(estimators=[('lm',
LinearRegression()),('dt',DecisionTreeRegressor())])),
            ("Vote", VotingRegressor(estimators=[('lm', LinearRegression()),
('dt',DecisionTreeRegressor())])),
            ("BoostStack", StackingRegressor(estimators=[('lm',
LinearRegression()),('rf',RandomForestRegressor()),
('xgb',xgb.XGBRegressor())])),
            ("SVM_l", SVR(kernel="linear")),
            ("SVM_p", SVR(kernel="poly")),
            ("SVM_r", SVR(kernel="rbf")),
            ("SVMStack", StackingRegressor(estimators=[('lm',
LinearRegression()),('svm',SVR(kernel="linear")),
('rf',RandomForestRegressor()),('xgb',xgb.XGBRegressor())])),]

resdf=[]
for mtype,model in reg_models:
    starttime=time.time()
    model.fit(egywth_train[["SDK", "NM", "VPM", "TMK", "Weekday"]],
egywth_train.ES_Lab.values)
    fittime=time.time()
    pred_train = model.predict(egywth_train[["SDK", "NM", "VPM", "TMK",
"Weekday"]])
    pred_test  = model.predict(egywth_test[["SDK", "NM", "VPM", "TMK",
"Weekday"]])
    predtime=time.time()
    resdf.append({"type":mtype,
                  "train.MSE":mean_squared_error(egywth_train.ES_Lab.values,
pred_train),

"train.RMSE":root_mean_squared_error(egywth_train.ES_Lab.values, pred_train),
                  "train.R2":r2_score(egywth_train.ES_Lab.values, pred_train),
                  "test.MSE":mean_squared_error(egywth_test.ES_Lab.values,
pred_test),

"test.RMSE":root_mean_squared_error(egywth_test.ES_Lab.values, pred_test),
                  "test.R2":r2_score(egywth_test.ES_Lab.values, pred_test),
                  "fittime":fittime-starttime,
                  "predtime":predtime-fittime})
```

```
pd.DataFrame(resdf)
```

	type	train.MSE	train.RMSE	train.R2	test.MSE	test.RMSE	test.R2	fittime	predtime
0	Linear Model	160.535	12.670	0.868	135.032	11.620	0.878	0.001	0.001
1	Decision Tree	0.000	0.000	1.000	229.763	15.158	0.793	0.002	0.001
2	Random Forest	20.043	4.477	0.983	129.328	11.372	0.883	0.081	0.012
3	Gradient Boost	62.037	7.876	0.949	123.967	11.134	0.888	0.036	0.002
4	XGBoost	0.153	0.391	1.000	149.610	12.232	0.865	0.185	0.002
5	Stack	111.975	10.582	0.908	125.673	11.210	0.887	0.014	0.002
6	Vote	40.134	6.335	0.967	137.036	11.706	0.876	0.002	0.001
7	BoostStack	53.125	7.289	0.956	119.286	10.922	0.892	1.557	0.015
8	SVM_l	163.019	12.768	0.866	133.849	11.569	0.879	0.008	0.004
9	SVM_p	214.008	14.629	0.824	176.615	13.290	0.841	0.006	0.004
10	SVM_r	250.421	15.825	0.794	211.702	14.550	0.809	0.004	0.013
11	SVMStack	51.329	7.164	0.958	121.138	11.006	0.891	1.606	0.018

Hier sehen wir, wieder das Overfitten der Entscheidungsbäume im Trainingsdatensatz, aber schlechter als Regressionsmodelle abschneiden im Testdatensatz. Wir sehen auch, dass die Regressions-Ensamble-Modelle sogar besser abschneiden als die klassischen linearen Regressionsmodelle. Das liegt daran das die zugrundeliegenden Entscheidungsbäume vom Normalen linearen Verhalten abweichende Ausnahmen besser erfassen können. Die Stacking Varianten schneiden zum Teil am besten ab, aber weisen auch die größten Trainingszeiten auf.

Bibliographie