KAPITEL 6

Entscheidungsbäume

Wie SVMs sind auch *Entscheidungsbäume* sehr flexible Machine-Learning-Algorithmen, die sich sowohl für Klassifikations- als auch Regressionsaufgaben eignen. Sogar Aufgaben mit multiplen Ausgaben lassen sich mit ihnen lösen. Es sind sehr mächtige Algorithmen, mit denen sich komplexe Datensätze fitten lassen. Beispielsweise haben Sie in Kapitel 2 ein Modell mit dem `DecisionTreeRegressor` trainiert und an den Datensatz zu kalifornischen Immobilien perfekt angepasst (genauer gesagt, overfittet).

Entscheidungsbäume sind außerdem die funktionelle Komponente von Random Forests (siehe Kapitel 7), die zu den mächtigsten heute verfügbaren Machine-Learning-Algorithmen gehören.

In diesem Kapitel werden wir besprechen, wie sich Entscheidungsbäume trainieren, visualisieren und für Vorhersagen einsetzen lassen. Anschließend werden wir den in Scikit-Learn verwendeten CART-Trainingsalgorithmus durchgehen. Wir werden betrachten, wie sich Entscheidungsbäume regularisieren und für Regressionsaufgaben einsetzen lassen. Schließlich werden wir einige Einschränkungen von Entscheidungsbäumen kennenlernen.

Trainieren und Visualisieren eines Entscheidungsbaums

Um Entscheidungsbäume zu verstehen, werden wir zunächst einen erstellen und uns ansehen, wie dieser Vorhersagen trifft. Der folgende Code trainiert einen `DecisionTreeClassifier` auf dem Iris-Datensatz (siehe Kapitel 4):

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # Länge und Breite der Kronblätter
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

Sie können den trainierten Entscheidungsbaum visualisieren, indem Sie durch Aufrufen der Methode `export_graphviz()` eine Datei namens *iris_tree.dot* mit einer Repräsentation als Graph erzeugen:

```
from sklearn.tree import export_graphviz

export_graphviz(
        tree_clf,
        out_file=image_path("iris_tree.dot"),
        feature_names=iris.feature_names[2:],
        class_names=iris.target_names,
        rounded=True,
        filled=True
    )
```

Diese *.dot*-Datei lässt sich anschließend mit dem Kommandozeilenprogramm `dot` aus dem Paket *graphviz* in verschiedene Formate wie PDF oder PNG umwandeln.[1] Der folgende Konsolenbefehl wandelt die *.dot*-Datei in ein *.png*-Bild um:

```
$ dot -Tpng iris_tree.dot -o iris_tree.png
```

Ihr erster Entscheidungsbaum ist der in Abbildung 6-1 dargestellte.

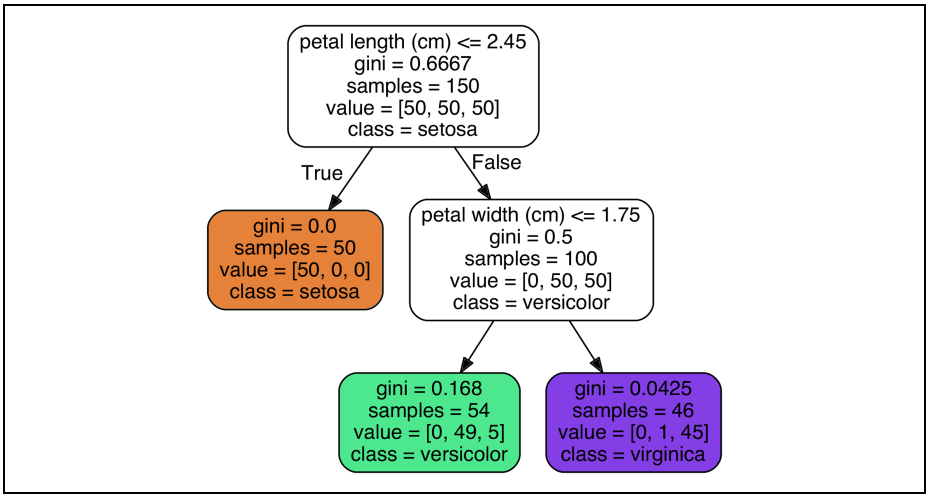

Abbildung 6-1: Iris-Entscheidungsbaum

Vorhersagen treffen

Betrachten wir nun, wie der in Abbildung 6-1 dargestellte Baum Vorhersagen trifft. Nehmen wir an, Sie finden eine Iris-Blüte und möchten diese klassifizieren. Sie beginnen an der *Wurzel* des Baums (bei depth 0, ganz oben): Dieser Knoten stellt

1 Graphviz ist ein Open-Source-Softwarepaket zur Visualisierung von Graphen und unter *http://www.graphviz.org/* verfügbar.

die Frage, ob das Kronblatt der Blüte kürzer als 2.45 cm ist. Ist dies der Fall, fahren Sie mit dem Kind auf der linken Seite fort (depth 1, links). In diesem Fall ist der Knoten ein *Blatt* (es besitzt keine weiteren Kinder) und stellt keine weiteren Fragen: Sie übernehmen einfach die von diesem Knoten vorhergesagte Kategorie, somit sagt unser Entscheidungsbaum vorher, dass Ihre Blüte eine Iris-Setosa (`class=setosa`) ist.

Nehmen wir an, Sie finden eine weiter Blüte, bei der das Kronblatt diesmal länger als 2.45 cm ist. Sie fahren mit dem rechten Kind der Wurzel fort (bei depth 1, rechts). Dieser Knoten ist kein Blatt, sondern stellt eine weitere Frage: Ist das Kronblatt schmaler als 1.75 cm? Ist dies der Fall, ist Ihre Blüte vermutlich eine Iris-Versicolor (bei depth 2, links). Wenn nicht, ist sie vermutlich eine Iris-Virginica (bei depth 2, rechts). Es ist tatsächlich so einfach.

Einer der vielen Vorzüge von Entscheidungsbäumen ist, dass sie sehr wenig Vorbereitung der Daten erfordern. Insbesondere ist keinerlei Skalierung oder Zentrierung von Merkmalen notwendig.

Das Attribut `samples` eines Knotens zählt, für wie viele Trainingsdatenpunkte dieser gültig ist. Beispielsweise haben 100 Datenpunkte ein Kronblatt mit einer Länge von mindestens 2.45 cm (bei depth 1, rechts), davon haben 54 ein Kronblatt mit einer Breite von weniger als 1.75 cm (bei depth 2, links). Das Attribut `value` eines Knotens verrät uns, wie viele Trainingsdatenpunkte der Knoten in jeder Kategorie enthält: Beispielsweise enthält der Knoten rechts unten 0 Exemplare von Iris-Setosa, 1 Iris-Versicolor und 45 Iris-Virginica. Schließlich misst das Attribut `gini` die *Unreinheit* eines Knotens: Ein Knoten gilt als »rein« (`gini=0`), wenn sämtliche enthaltenen Datenpunkte der gleichen Kategorie angehören. Da beispielsweise der linke Knoten bei depth-1 ausschließlich Instanzen von Iris-Setosa enthält, ist er rein und besitzt einen `gini`-Score von 0. Formel 6-1 stellt dar, wie der Trainingsalgorithmus den `gini`-Score G_i für den i^{ten} Knoten berechnet. Beispielsweise ist der gini-Score für den linken Knoten bei depth-2 gleich $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$. Wir werden in Kürze ein weiteres Maß für *Unreinheit* kennenlernen.

Formel 6-1: Gini-Unreinheit

$$G_i = 1 - \sum_{k=1}^{n} {p_{i,k}}^2$$

- $p_{i,k}$ ist dabei der Anteil von Instanzen der Kategorie k an den Datenpunkten im Knoten i.

Scikit-Learn verwendet den CART-Algorithmus, der ausschließlich *Binärbäume* erzeugt: Innere Knoten haben stets zwei Kinder (d.h., die Fragen lassen sich nur mit Ja oder Nein beantworten). Es existieren aber auch andere Algorithmen wie ID3, die Entscheidungsbäume erzeugen, deren Knoten mehr als zwei Kinder haben können.

Abbildung 6-2 zeigt die Entscheidungsgrenzen dieses Entscheidungsbaums. Die dicke vertikale Linie steht für die Entscheidungsgrenze der Wurzel (depth 0): petal length = 2.45 cm. Da das Gebiet auf der linken Seite rein ist (ausschließlich Iris-Setosa), lässt es sich nicht weiter aufteilen. Das Gebiet auf der rechten Seite ist dagegen unrein, daher teilt es der Knoten bei depth-1 auf der Höhe petal width = 1.75 cm auf (als gestrichelte Linie dargestellt). Da `max_depth` auf 2 gesetzt wurde, endet der Entscheidungsbaum an dieser Stelle. Wenn Sie allerdings `max_depth` auf 3 setzen, würden die zwei Knoten bei depth-2 jeweils eine zusätzliche Entscheidungsgrenze produzieren (hier durch die gepunkteten Linien dargestellt).

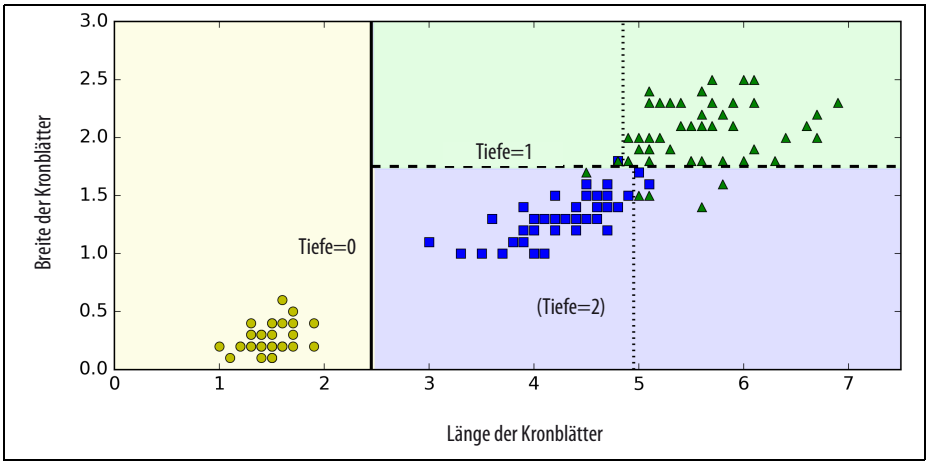

Abbildung 6-2: Entscheidungsgrenzen in einem Entscheidungsbaum

Interpretation von Modellen: White Box im Vergleich zu Blackbox

Wie Sie sehen, sind Entscheidungsbäume recht einfach nachzuvollziehen und ihre Entscheidungen leicht interpretierbar. Solche Modelle werden oft als *White-Box-Modelle* bezeichnet. Im Gegensatz dazu gehören, wie wir noch sehen werden, Random Forests oder neuronale Netze im Allgemeinen zu den *Blackbox-Modellen*. Sie treffen zwar ausgezeichnete Vorhersagen, und Sie können die dazu durchgeführten Berechnungen leicht nachprüfen; trotzdem ist es in der Regel schwierig, in wenigen Worten zu erklären, warum die Vorhersagen in einer bestimmten Weise getroffen wurden. Wenn beispielsweise ein neuronales Netzwerk behauptet, dass eine bestimmte Person auf einem Bild zu sehen ist, kann man nur schwer erkennen, worauf sich diese Vorhersage stützt: Hat das Modell die Augen der Person erkannt? Den Mund? Die Nase? Die Schuhe? Oder gar das Sofa, auf dem die Person saß? Umgekehrt geben uns Entscheidungsbäume einfache, klare Regeln zur Klassifikation, die notfalls sogar von Hand durchgeführt werden können (z. B. bei der Klassifikation von Blüten).

Schätzen von Wahrscheinlichkeiten für Kategorien

Ein Entscheidungsbaum kann auch die Wahrscheinlichkeit für die Zugehörigkeit eines Datenpunkts zu einer Kategorie *k* abschätzen: Zuerst schreitet das Verfahren den Baum ab, um das Blatt für diesen Datenpunkt zu finden, und gibt dann den Anteil der Trainingsdatenpunkte der Kategorie *k* in diesem Knoten zurück. Nehmen wir an, Sie hätten eine Blüte mit 5 cm langen und 1.5 cm breiten Kronblättern entdeckt. Der dazu passende Knoten im Entscheidungsbaum ist der Knoten bei depth-2 auf der linken Seite, daher sollte der Entscheidungsbaum die folgenden Wahrscheinlichkeiten ausgeben: 0% für Iris-Setosa (0/54), 90.7% für Iris-Versicolor (49/54) und 9.3% für Iris-Virginica (5/54). Wenn Sie nach einer Vorhersage der Kategorie fragen, erhalten Sie natürlich Iris-Versicolor (Kategorie 1), da diese die höchste Wahrscheinlichkeit hat. Überprüfen wir dies:

```
>>> tree_clf.predict_proba([[5, 1.5]])
array([[ 0. ,  0.90740741,  0.09259259]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

Perfekt! Beachten Sie, dass die geschätzten Wahrscheinlichkeiten überall in der unteren rechten Ecke von Abbildung 6-2 die gleichen sind – sogar wenn die Kronblätter beispielsweise 6 cm lang und 1.5 cm breit wären (obwohl es sich dann höchstwahrscheinlich um eine Iris-Virginica handeln würde).

Der CART-Trainings-Algorithmus

Scikit-Learn verwendet den *Classification and Regression Tree*-Algorithmus (CART-Algorithmus), um Entscheidungsbäume zu trainieren (oder »anzubauen«). Er folgt einer sehr einfachen Grundidee: Der Algorithmus teilt die Trainingsdaten zunächst anhand eines Merkmals *k* und eines Schwellenwerts t_k in zwei Untermengen auf (z.B. »petal length ≤ 2.45 cm«). Wie werden *k* und t_k ausgewählt? Der Algorithmus sucht nach dem Paar (k, t_k), das die reinsten (nach deren Größe gewichteten) Untermengen hervorbringt. Dabei versucht der Algorithmus, die Kostenfunktion in Formel 6-2 zu minimieren.

Formel 6-2: Kostenfunktion des CART-Algorithmus zur Klassifikation

$$J(k, t_k) = \frac{m_{\text{links}}}{m} G_{\text{links}} + \frac{m_{\text{rechts}}}{m} G_{\text{rechts}}$$

$$\text{wobei} \begin{cases} G_{\text{links/rechts}} \text{ die Unreinheit der linken/rechten Untermenge misst,} \\ m_{\text{links/rechts}} \text{ die Anzahl Datenpunkte in der linken/rechten Untermenge ist.} \end{cases}$$

Sobald der Trainingsdatensatz erfolgreich zweigeteilt wurde, werden die Untermengen nach dem gleichen Verfahren weiter aufgeteilt. Dies setzt sich rekursiv fort, bis die (durch den Hyperparameter `max_depth` angegebene) maximale Tiefe erreicht wurde oder keine Aufteilung gefunden werden kann, die die Unreinheit weiter

reduziert. Andere Hyperparameter steuern weitere Abbruchbedingungen (`min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf` und `max_leaf_nodes`). Diese erklären wir in Kürze.

Wie Sie sehen, gehört der CART-Algorithmus zu den Greedy-Algorithmen: Er sucht gierig nach einer optimalen Aufteilung auf der höchsten möglichen Ebene und setzt diesen Vorgang auf jeder Stufe fort. Es wird nicht geprüft, ob eine Aufteilung einige Schritte weiter zur höchsten möglichen Unreinheit führt. Ein Greedy-Algorithmus liefert oft eine recht gute Lösung, diese muss aber nicht die bestmögliche sein.

Leider gehört das Finden des optimalen Baums zu den *NP-vollständigen* Problemen:[2] Es erfordert eine Zeit von $O(\exp(m))$, wodurch das Problem selbst für eher kleine Datensätze praktisch unlösbar wird. Daher müssen wir uns mit einer »annehmbar guten« Lösung zufriedengeben.

Komplexität der Berechnung

Das Treffen von Vorhersagen erfordert das Abschreiten eines Entscheidungsbaums von der Wurzel zu einem Blatt. Entscheidungsbäume sind halbwegs ausbalanciert, daher erfordert das Abschreiten etwa $O(log_2(m))$ Knoten.[3] Da bei jedem Knoten nur ein Merkmal geprüft werden muss, ist die Komplexität der Vorhersage lediglich $O(log_2(m))$, egal wie viele Merkmale es gibt. Die Vorhersagen sind also auch für große Trainingsdatensätze sehr schnell.

Allerdings vergleicht der Trainingsalgorithmus bei jedem Knoten sämtliche Merkmale (falls `max_features` gesetzt ist auch weniger) aller Datenpunkte miteinander. Dies führt zu einer Komplexität von $O(n \times m\ log(m))$ beim Trainieren. Bei kleinen Trainingsdatensätzen (weniger als einige Tausend Datenpunkte) kann Scikit-Learn das Trainieren beschleunigen, indem es die Daten vorsortiert (mit `presort=True`). Bei größeren Trainingsdatensätzen verlangsamt dies das Trainieren aber deutlich.

2 P ist die Menge der in polynomieller Zeit lösbaren Probleme. NP ist die Menge der Probleme, deren Lösungen sich in polynomieller Zeit überprüfen lassen. Ein NP-schweres Problem ist ein Problem, zu dem sich jedes NP-Problem in polynomieller Zeit reduzieren lässt. Ein NP-vollständiges Problem ist sowohl NP als auch NP-schwer. Ob P = NP gilt, ist eine wichtige mathematische Frage. Falls $P \neq NP$ gilt (was wahrscheinlich erscheint), kann es für kein NP-vollständiges Problem jemals einen polynomiellen Algorithmus geben (außer vielleicht auf einem Quantencomputer).

3 $\_log\__2$ ist der binäre Logarithmus. Er entspricht $log_2(m) = log(m) / log(2)$.

Gini-Unreinheit oder Entropie?

Standardmäßig wird die Gini-Unreinheit verwendet, Sie können aber stattdessen auch die *Entropie* als Maß für die Unreinheit auswählen, indem Sie den Hyperparameter `criterion` auf `"entropy"` setzen. Der aus der Thermodynamik stammende Begriff Entropie beschreibt Unordnung auf molekularer Ebene: Die Entropie nähert sich null, wenn Moleküle unbeweglich und wohlgeordnet sind. Dieses Konzept hat sich später in andere Fachgebiete ausgebreitet, darunter Shannons *Informationstheorie*, wo sie den durchschnittlichen Informationsgehalt einer Nachricht misst:[4] Wenn alle Nachrichten identisch sind, beträgt die Entropie null. Im Machine Learning wird die Entropie oft als Maß für die Unreinheit eingesetzt: Die Entropie einer Menge ist null, wenn Sie nur aus Datenpunkten einer Kategorie besteht. Formel 6-3 zeigt die Definition der Entropie des i^{ten} Knotens. Beispielsweise beträgt die Entropie für den linken Knoten bei depth-2 in Abbildung 6-1 genau $-\frac{49}{54}\log\left(\frac{49}{54}\right) - \frac{5}{54}\log\left(\frac{5}{54}\right) \approx 0.31$.

Formel 6-3: Entropie

$$H_i = -\sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^{n} p_{i,k} \log\left(p_{i,k}\right)$$

Sollten Sie also die Gini-Unreinheit oder die Entropie verwenden? Tatsächlich macht es meist keinen großen Unterschied: Beide ergeben ähnliche Bäume. Die Gini-Unreinheit lässt sich ein wenig schneller berechnen und eignet sich daher als Standardwert. Wenn beide Maße voneinander abweichen, neigt die Gini-Unreinheit dazu, die häufigste Kategorie in einem eigenen Ast des Baums abzusondern, wohingegen die Entropie etwas ausbalanciertere Bäume erzeugt.[5]

Hyperparameter zur Regularisierung

Entscheidungsbäume treffen sehr wenige Annahmen über die Trainingsdaten (im Gegensatz zu beispielsweise linearen Modellen, die offensichtlich annehmen, dass sich die Daten linear verhalten). Sich selbst überlassen, passt sich die Struktur des Baums sehr genau an die Trainingsdaten an und führt höchstwahrscheinlich zu Overfitting. Solch ein Modell wird auch als *parameterfreies Modell* bezeichnet, nicht weil es keine Parameter gäbe (meist gibt es viele), sondern weil die Anzahl der Parameter nicht vor dem Trainieren festgelegt wird. Daher ist es dem Modell über-

4 Ein Reduzieren der Entropie wird oft als *Zugewinn an Information* bezeichnet.

5 Details finden Sie in einer interessanten Analyse von Sebastian Raschka (*http://goo.gl/UndTrO*).

lassen, sich eng an die Daten anzupassen. Im Gegensatz dazu besitzt ein *parametrisches Modell* wie etwa ein lineares Modell eine im Voraus festgelegte Anzahl Parameter. Es verfügt daher über eine begrenzte Anzahl Freiheitsgrade, wodurch es weniger zu Overfitting neigt (aber dafür ein höheres Risiko für Underfitting besteht).

Um ein Overfitting der Trainingsdaten zu vermeiden, müssen Sie die Freiheitsgrade eines Entscheidungsbaums beim Trainieren einschränken. Wie Sie inzwischen wissen, nennt man dies Regularisierung. Die Hyperparameter zur Regularisierung hängen vom verwendeten Algorithmus ab. Im Allgemeinen können Sie aber mindestens die maximale Tiefe des Entscheidungsbaums begrenzen. In Scikit-Learn lässt sich dies über den Hyperparameter `max_depth` erreichen (die Voreinstellung ist `None`, eine unbegrenzte Tiefe). Ein Reduzieren von `max_depth` regularisiert das Modell und verringert damit das Risiko einer Überanpassung.

Die Klasse `DecisionTreeClassifier` bietet einige weitere Parameter, die die Form des Entscheidungsbaums in ähnlicher Weise einschränken: `min_samples_split` (die minimale Anzahl von Datenpunkten, die ein Knoten aufweisen muss, damit er aufgeteilt werden kann), `min_samples_leaf` (die minimale Anzahl Datenpunkte, die ein Blatt haben muss), `min_weight_fraction_leaf` (wie `min_samples_leaf`, aber als Anteil der gesamten gewichteten Datenpunkte), `max_leaf_nodes` (maximale Anzahl Blätter) und `max_features` (maximale Anzahl beim Aufteilen eines Knotens berücksichtigter Merkmale). Ein Erhöhen der `min_*`-Hyperparameter und ein Senken der `max_*`-Hyperparameter regularisiert das Modell.

Andere Algorithmen trainieren Entscheidungsbäume zunächst ohne Einschränkungen, entfernen aber anschließend überflüssige Knoten (*Pruning*). Ein Knoten wird als überflüssig angesehen, wenn seine Kinder ausschließlich Blätter sind und der durch ihn erbrachte Zugewinn an Reinheit nicht *statistisch signifikant* ist. Standardisierte statistische Tests wie der χ^2-*Test* schätzen die Wahrscheinlichkeit ab, dass eine Verbesserung rein zufällig erfolgt ist (dies bezeichnet man als die *Nullhypothese*). Wenn diese Wahrscheinlichkeit, genannt *p-Wert*, höher als ein eingestellter Schwellenwert ist (typischerweise 5 %, durch einen Hyperparameter steuerbar), dann wird der Knoten als überflüssig erachtet und seine Kinder gelöscht. Dieses Pruning wird fortgesetzt, bis alle überflüssigen Knoten entfernt worden sind.

Abbildung 6-3 zeigt zwei auf dem Datensatz moons (aus Kapitel 5) trainierten Entscheidungsbäume. Der Entscheidungsbaum auf der linken Seite wurde mit den voreingestellten Hyperparametern trainiert (also ohne Restriktionen), der auf der rechten Seite mit `min_samples_leaf=4`. Es wird schnell deutlich, dass das Modell auf der linken Seite overfittet ist und das Modell auf der rechten Seite vermutlich besser verallgemeinert.

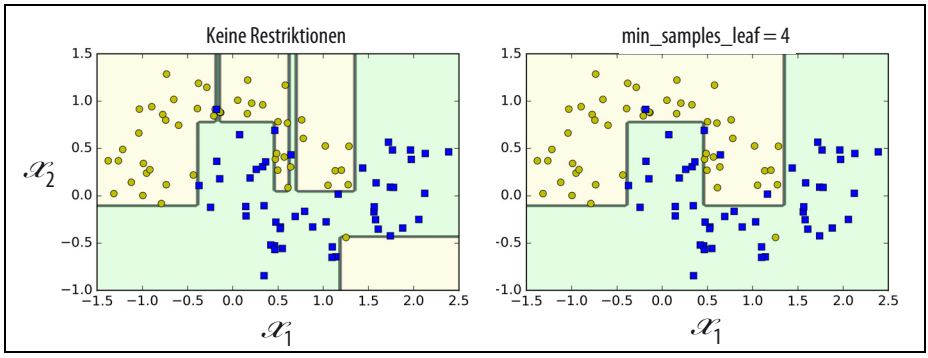

Abbildung 6-3: Regularisierung mit min_samples_leaf

Regression

Entscheidungsbäume können auch Regressionsaufgaben bewältigen. Erstellen wir mit der Klasse `DecisionTreeRegressor` aus Scikit-Learn einen Regressionsbaum und trainieren diesen auf einem verrauschten quadratischen Datensatz mit `max_depth=2`:

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(max_depth=2)
tree_reg.fit(X, y)
```

Der dabei erhaltene Baum ist in Abbildung 6-4 dargestellt.

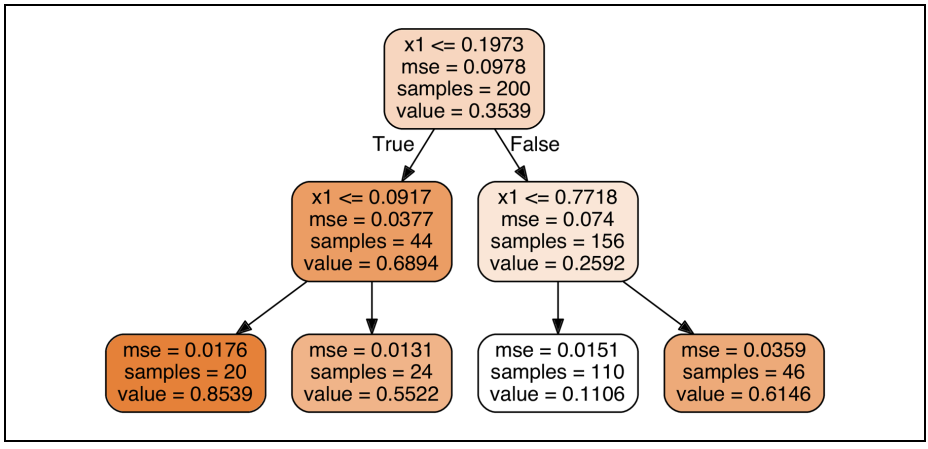

Abbildung 6-4: Ein Entscheidungsbaum zur Regression

Dieser Baum sieht dem zuvor zur Klassifikation erstellten Baum sehr ähnlich. Der Hauptunterschied ist, dass er anstelle einer Kategorie in jedem Knoten einen Wert vorhersagt. Wenn Sie beispielsweise eine Vorhersage für einen neuen Datenpunkt

bei $x_1 = 0.6$ treffen möchten, schreiten Sie den Baum von der Wurzel ausgehend ab. Sie erreichen irgendwann das Blatt mit der Vorhersage `value=0.1106`. Diese Vorhersage ist nichts weiter als der durchschnittliche Zielwert der 110 Trainingsdatenpunkte in diesem Blatt. Diese Vorhersage führt zu einem mittleren quadratischen Fehler (MSE) von 0.0151 in diesen 110 Datenpunkten.

Die Vorhersagen dieses Modells sind auf der linken Seite von Abbildung 6-5 dargestellt. Wenn Sie `max_depth=3` setzen, erhalten Sie die auf der rechten Seite dargestellten Vorhersagen. Beachten Sie, dass der vorhergesagte Wert in jedem Abschnitt dem durchschnittlichen Zielwert der Datenpunkte in diesem Abschnitt entspricht. Der Algorithmus teilt jeden Abschnitt so auf, dass möglichst viele Trainingsdatenpunkte so nah wie möglich am vorhergesagten Wert liegen.

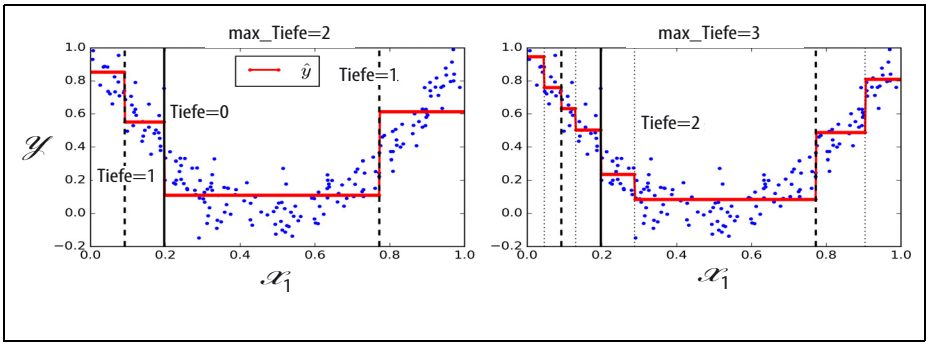

Abbildung 6-5: Vorhersagen zweier Entscheidungsbäume zur Regression

Der CART-Algorithmus funktioniert weitgehend wie oben vorgestellt. Der einzige Unterschied besteht darin, dass der Trainingsdatensatz so aufgeteilt wird, dass der MSE anstatt der Unreinheit minimiert wird. Formel 6-4 zeigt die vom Algorithmus minimierte Kostenfunktion.

Formel 6-4: CART-Kostenfunktion für die Regression

$$J(k, t_k) = \frac{m_{\text{left}}}{m}\text{MSE}_{\text{links}} + \frac{m_{\text{rechts}}}{m}\text{MSE}_{\text{rechts}} \quad \text{wobei} \begin{cases} \text{MSE}_{\text{Knoten}} = \sum_{i \in \text{Knoten}} (\hat{y}_{\text{Knoten}} - y^{(i)})^2 \\ \hat{y}_{\text{Knoten}} = \frac{1}{m_{\text{Knoten}}} \sum_{i \in \text{Knoten}} y^{(i)} \end{cases}$$

Wie bei Klassifikationsaufgaben sind auch Entscheidungsbäume für Regressionsaufgaben anfällig für Overfitting. Ohne jegliche Regularisierung (z. B. mit den voreingestellten Hyperparametern) erhalten Sie die Vorhersagen auf der linken Seite von Abbildung 6-6. Dies ist offensichtlich ein schwerwiegendes Overfitting der Trainingsdaten. Durch Setzen von `min_samples_leaf=10` erhalten Sie ein weitaus sinnvolleres Modell, das Sie auf der rechten Seite von Abbildung 6-6 sehen.

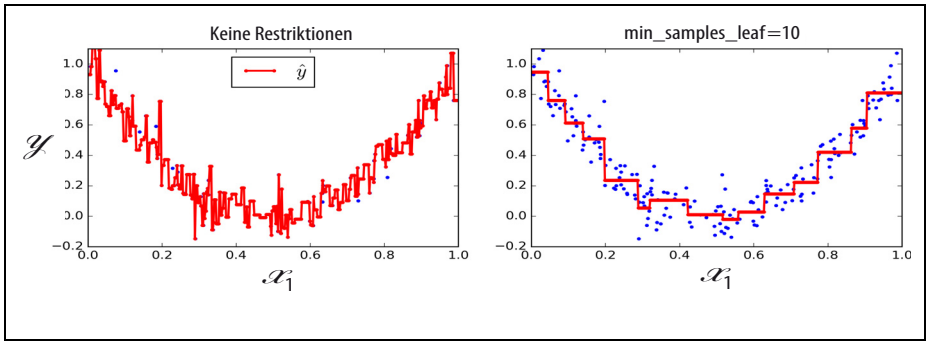

Abbildung 6-6: Regularisierung eines Regressionsbaums

Instabilität

Sie sind hoffentlich mittlerweile davon überzeugt, dass sehr vieles für Entscheidungsbäume spricht: Sie sind einfach zu verstehen und zu interpretieren, leicht anwendbar und mächtig. Sie haben allerdings auch einige Schwachstellen: Erstens haben Sie womöglich bereits festgestellt, dass Entscheidungsbäume orthogonale Entscheidungsgrenzen lieben (alle Unterteilungen stehen im rechten Winkel zu einer Achse). Dadurch reagieren sie empfindlich auf Rotationen der Trainingsdaten. Als Beispiel zeigt Abbildung 6-7 einen einfachen linear separierbaren Datensatz: Auf der linken Seite kann der Entscheidungsbaum diesen einfach unterteilen. Auf der rechten Seite wirkt die Entscheidungsgrenze nach einer Drehung des Datensatzes um 45° unnötig verschachtelt. Obwohl beide Entscheidungsbäume die Trainingsdaten perfekt abbilden, verallgemeinert das Modell auf der rechten Seite vermutlich nicht besonders gut. Dieses Problem lässt sich über eine Hauptkomponentenzerlegung angehen (siehe Kapitel 8), die häufig zu einer besseren Ausrichtung der Trainingsdaten führt.

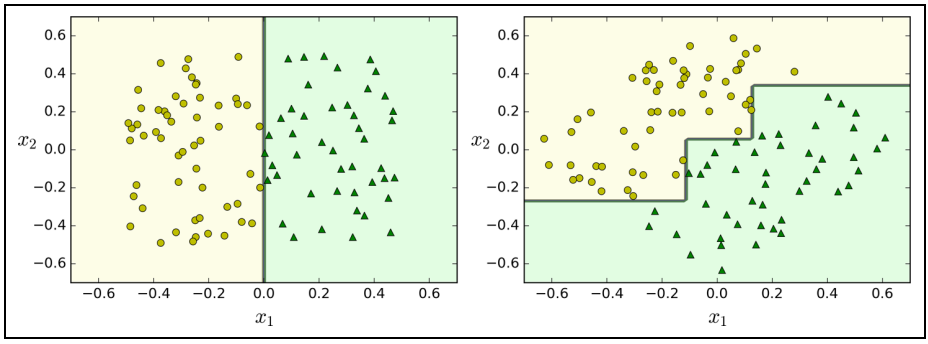

Abbildung 6-7: Empfindlichkeit für Rotation der Trainingsdaten

Allgemeiner gesprochen ist das Hauptproblem bei Entscheidungsbäumen, dass sie sehr empfindlich auf kleine Variationen in den Trainingsdaten reagieren. Wenn Sie beispielsweise einfach die breiteste Iris-Versicolor aus dem Iris-Trainingsdatensatz entfernen (diejenige mit 4.8 cm langen und 1.8 cm breiten Kronblättern) und einen neuen Entscheidungsbaum trainieren, erhalten Sie das in Abbildung 6-8 gezeigte Modell. Wie Sie sehen, unterscheidet es sich sehr stark vom vorigen Entscheidungsbaum (Abbildung 6-2). Genauer gesagt, da der von Scikit-Learn verwendete Trainingsalgorithmus stochastisch[6] arbeitet, können Sie sogar mit den gleichen Trainingsdaten sehr unterschiedliche Modelle erhalten (es sei denn, Sie setzen den Hyperparameter `random_state`).

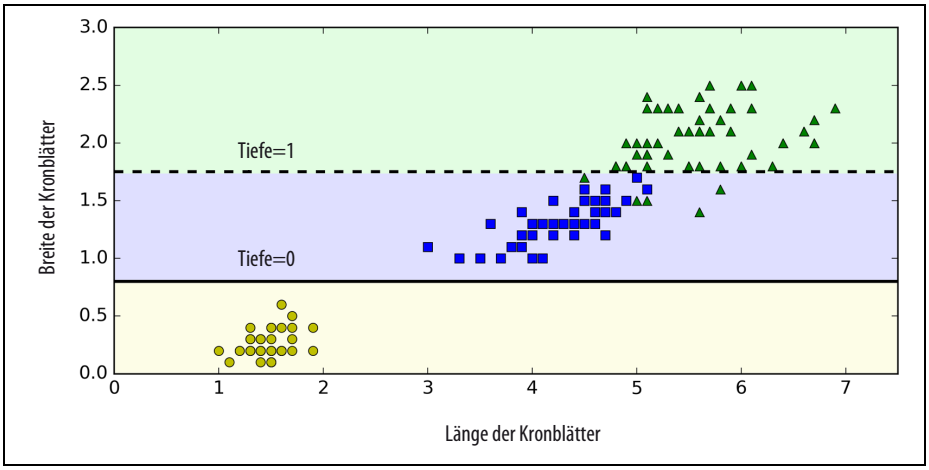

Abbildung 6-8: Anfälligkeit für Einzelheiten der Trainingsdaten

Random Forests wirken dieser Instabilität entgegen, indem sie die Vorhersagen vieler Bäume mitteln, wie wir im nächsten Kapitel sehen werden.

Übungen

1. Was ist die ungefähre Tiefe eines mit 1 Million Datenpunkten (ohne Restriktionen) trainierten Entscheidungsbaums?
2. Ist die Gini-Unreinheit eines Knotens im Allgemeinen geringer oder größer als die seines Elternteils? Ist sie *im Allgemeinen* kleiner/größer oder *immer* kleiner/größer?
3. Sollte man versuchen, `max_depth` zu senken, wenn ein Entscheidungsbaum einen Trainingsdatensatz overfittet?

6 Die bei jedem Knoten zu evaluierenden Merkmale werden zufällig ausgewählt.

4. Sollte man versuchen, die Eingabemerkmale zu skalieren, wenn ein Entscheidungsbaum die Trainingsdaten underfittet?
5. Wenn es eine Stunde dauert, einen Entscheidungsbaum mit 1 Million Datenpunkten zu trainieren, wie lange etwa wird das Trainieren eines weiteren Baums mit 10 Millionen Datenpunkten dauern?
6. Wenn Ihr Trainingsdatensatz aus 100000 Datenpunkten besteht, beschleunigt das Setzen von `presort=True` das Trainieren?
7. Trainieren und optimieren Sie einen Entscheidungsbaum für den Datensatz moons.
    a. Erzeugen Sie einen moons-Datensatz mit `make_moons(n_samples=10000, noise=0.4)`.
    b. Teilen Sie ihn mit `train_test_split()` in einen Trainings- und einen Testdatensatz auf.
    c. Verwenden Sie die Gittersuche mit Kreuzvalidierung (mithilfe der Klasse `GridSearchCV`), um gute Einstellungen für die Hyperparameter eines `DecisionTreeClassifier` zu finden. Hinweis: Probieren Sie unterschiedliche Werte für `max_leaf_nodes`.
    d. Trainieren Sie den Baum mit den vollständigen Trainingsdaten und bestimmen Sie die Qualität Ihres Modells auf den Testdaten. Sie sollten eine Genauigkeit zwischen 85 % und 87 % erhalten.
8. Züchten Sie einen Wald.
    a. Erzeugen Sie im Anschluss an die vorige Aufgabe 1000 Untermengen Ihres Trainingsdatensatzes mit jeweils 100 zufällig ausgewählten Datenpunkten. Hinweis: Sie können dazu die Klasse `ShuffleSplit` aus Scikit-Learn verwenden.
    b. Trainieren Sie auf jedem der Teildatensätze einen Entscheidungsbaum mit den besten oben gefundenen Hyperparametern. Werten Sie diese 1000 Entscheidungsbäume auf den Testdaten aus. Da sie mit kleineren Datensätzen trainiert wurden, schneiden diese Bäume mit nur ca. 80 % voraussichtlich schlechter als der erste Entscheidungsbaum ab.
    c. Nun kommt der Zaubertrick. Generieren Sie für jeden Testdatenpunkt die Vorhersagen der 1000 Entscheidungsbäume und heben Sie ausschließlich die häufigste Vorhersage auf (Sie können dazu die Funktion `mode()` aus SciPy verwenden). Damit erhalten Sie *Mehrheitsvorhersagen* für Ihren Testdatensatz.
    d. Werten Sie diese Vorhersagen mit dem Testdatensatz aus: Sie sollten eine etwas höhere Genauigkeit als beim ersten Modell erhalten (etwa 0.5 bis 1.5 % höher). Herzlichen Glückwunsch, Sie haben soeben einen Random-Forest-Klassifikator trainiert!

Lösungen zu diesen Übungen finden Sie in Anhang A.