Лабораторная работа №9

Ансамбли

Ансамбли *(ensembles)* – это методы, которые сочетают в себе множество моделей машинного обучения, чтобы в итоге получить более мощную модель. Существует много моделей машинного обучения, которые принадлежат к этой категории, но есть две ансамблевых модели, которые доказали свою эффективность на самых различных наборах данных для задач классификации и регрессии, обе используют деревья решений в качестве строительных блоков: случайный лес деревьев решений и градиентный бустинг деревьев решений.

Случайный лес

Как мы только что отметили, основным недостатком деревьев решений является их склонность к переобучению. Случайный лес является одним из способов решения этой проблемы. По сути случайный лес – это набор деревьев решений, где каждое дерево немного отличается от остальных. Идея случайного леса заключается в том, что каждое дерево может довольно хорошо прогнозировать, но скорее всего переобучается на части данных. Если мы построим много деревьев, которые хорошо работают и переобучаются с разной степенью, мы можем уменьшить переобучение путем усреднения их результатов. Уменьшение переобучения при сохранении прогнозной силы деревьев можно проиллюстрировать с помощью строгой математики.

Для реализации вышеизложенной стратегии нам нужно построить большое количество деревьев решений. Каждое дерево должно на приемлемом уровне прогнозировать целевую переменную и должно отличаться от других деревьев. Случайные леса получили свое название из-за того, что в процесс построения деревьев была внесена случайность, призванная обеспечить уникальность каждого дерева. Существует две техники, позволяющие получить рандомизированные деревья в рамках случайного леса: сначала выбираем точки данных (наблюдения), которые будут использоваться для построения дерева, а затем отбираем признаки в каждом разбиении. Давайте разберем этот процесс более подробно.

Построение случайного леса

Для построения модели случайных лесов необходимо определиться с количеством деревьев (параметр **n_estimators** для **RandomForestRegressor** или **RandomForestClassifier**). Допустим, мы хотим построить 10 деревьев. Эти деревья будут построены совершенно независимо друг от друга, и алгоритм будет случайным образом отбирать признаки для построения каждого дерева, чтобы получить непохожие друг на друга деревья. Для построения дерева мы сначала сформируем бутстреп-выборку (**bootstrap sample**) наших данных. То есть из **n_samples** примеров мы случайным образом выбираем пример с возвращением **n_samples** раз (поскольку отбор с возвращением, то один и тот же пример может быть выбран несколько раз). Мы получаем выборку, которая имеет такой же размер, что и исходный набор данных, однако некоторые примеры будут отсутствовать в нем (примерно одна треть), а некоторые попадут в него несколько раз.

Чтобы проиллюстрировать это, предположим, что мы хотим создать бутстреп-выборку списка ['a', 'b', 'c', 'd']. Возможная бутстреп-выборка может выглядеть как ['b', 'd', 'd', 'c']. Другой возможной бутстреп-выборкой может быть ['d', 'a', 'd', 'a'].

Далее на основе этой сформированной бутстреп-выборки строится дерево решений. Однако алгоритм, который мы описывали для дерева решений, теперь слегка изменен. Вместо поиска наилучшего теста для каждого узла, алгоритм для разбиения узла случайным образом отбирает подмножество признаков и затем находит наилучший тест, используя один из этих признаков. Количество отбираемых признаков контролируется параметром **max_features**. Отбор подмножества признаков повторяется отдельно для каждого узла, поэтому в каждом узле дерева может быть принято решение с использованием «своего» подмножества признаков.

Использование бутстрепа приводит к тому, что деревья решений в случайном лесе строятся на немного отличающихся между собой бутстреп-выборках. Из-за случайного отбора признаков в каждом узле все расщепления в деревьях будут основано на отличающихся подмножествах признаков. Вместе эти два механизма приводят к тому, что все деревья в случайном лесе отличаются друг от друга.

Критическим параметром в этом процессе является **max_features**. Если мы установим **max_features** равным **n_features**, это будет означать, что в каждом разбиении могут участвовать все признаки набора данных, в отбор признаков не будет привнесена случайность (впрочем, случайность в силу использования бутстрепа остается). Если мы установим **max_features** равным 1, это означает, что при разбиении не будет никакого отбора признаков для тестирования вообще, будет осуществляться поиск с учетом различных пороговых значений для случайно выбранного признака. Таким образом, высокое значение **max_features** означает, что деревья в случайном лесе будут весьма схожи между собой и они смогут легко аппроксимировать данные, используя наиболее дискриминирующие признаки. Низкое значение **max_features** означает, что деревья в случайном лесе будут сильно отличаться друг от друга и, возможно, каждое дерево будет иметь очень большую глубину, чтобы хорошо соответствовать данным.

Чтобы дать прогноз для случайного леса, алгоритм сначала дает прогноз для каждого дерева в лесе. Для регрессии мы можем усреднить эти результаты, чтобы получить наш окончательный прогноз. Для классификации используется стратегия «мягкого голосования». Это означает, что каждый алгоритм дает «мягкий» прогноз, вычисляя вероятности для каждого класса. Эти вероятности усредняются по всем деревьям и прогнозируется класс с наибольшей вероятностью.

Анализ случайного леса

Давайте применим случайный лес, состоящий из пяти деревьев, к набору данных two_moons, который мы изучали ранее:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(
 X, y, stratify=y, random_state=42
)
forest = RandomForestClassifier(n_estimators=5, random_state=2)
forest.fit(X_train, y_train)
```

Деревья, которые строятся в рамках случайного леса, сохраняются в атрибуте **estimator_.** Давайте визуализируем границы принятия решений, полученные каждым деревом, а затем выведем агрегированный прогноз, выданный лесом (рис. 9.1):

```
fig, axes = plt.subplots(2, 3, figsize=(20, 10))
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
    ax.set_title("Дерево {}".format(i))
    mglearn.plots.plot_tree_partition(X_train, y_train, tree, ax=ax)
    mglearn.plots.plot_2d_separator(
        forest, X_train, fill=True, ax=axes[-1, -1], alpha=.4
    )

axes[-1, -1].set_title("Случайный лес")
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
```

На рисунках отчетливо видно, что границы принятия решений, полученные с помощью пяти деревьев, существенно различаются между собой. Каждое дерево совершает ряд ошибок, поскольку из-за бутстрепа некоторые точки исходного обучающего набора фактически не были включены в обучающие наборы, по которым строились деревья.

Отличие от отдельных деревьев случайный лес переобучается в меньшей степени и дает гораздо более чувствительную (гибкую) границу принятия решений. В реальных примерах используется гораздо большее количество деревьев (часто сотни или тысячи), что приводит к получению еще более чувствительной границы.

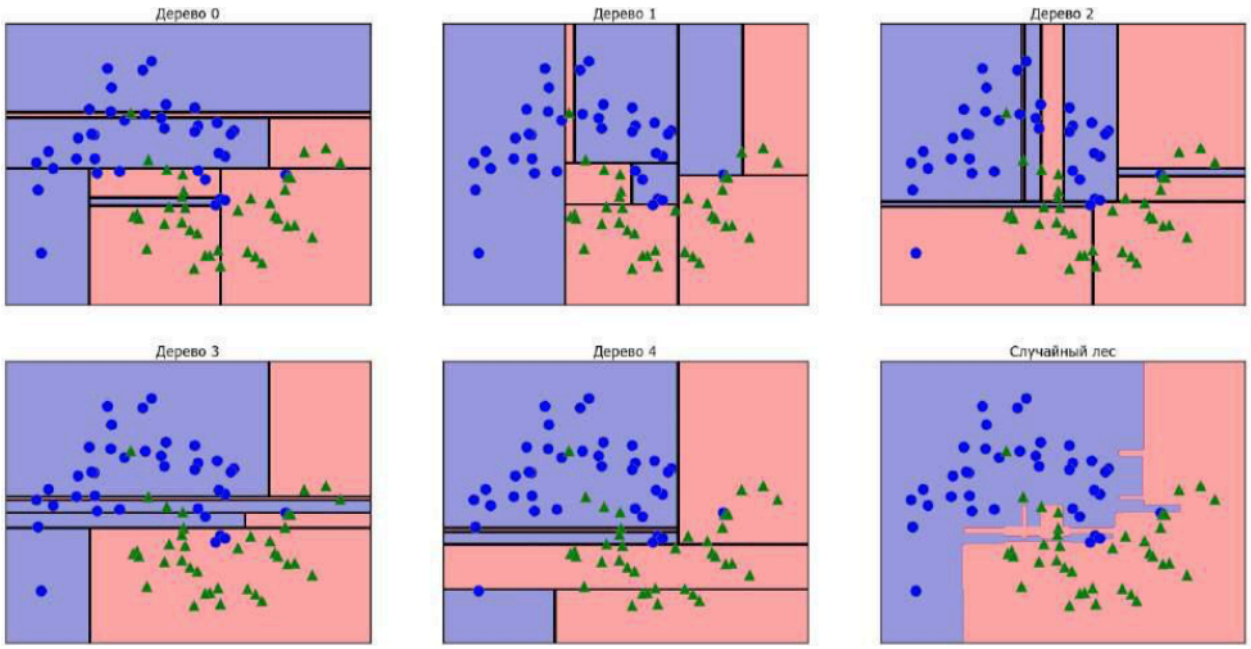

Рис. 9.1 Границы принятия решений, найденные пятью рандомизированными деревьями решений, и граница принятия решений, полученная путем усреднения их спрогнозированных вероятностей

В качестве еще одного примера давайте построим случайный лес, состоящий из 100 деревьев, на наборе данных Breast Cancer:

```
X_train, X_test, y_train, y_test = train_test_split(
  cancer.data,
  cancer.target,
  random_state=0
)
forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(X_train, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(
  forest.score(X_train, y_train))
)
print("Правильность на тестовом наборе: {:.3f}".format(
  forest.score(X_test, y_test))
)
```

```
Правильность на обучающем наборе: 1.000
Правильность на тестовом наборе: 0.972
```

Без настройки каких-либо параметров случайный лес дает нам правильность 97%, это лучше результата линейных моделей или одиночного дерева решений. Мы могли бы отрегулировать настройку **max_features** или применить предварительную обрезку, как это делали для одиночного дерева решений. Однако часто параметры случайного леса, выставленные по умолчанию, работают уже сами по себе достаточно хорошо.

Как и дерево решений, случайный лес позволят вычислить важности признаков, которые рассчитываются путем агрегирования значений важности по всем деревьям леса. Как правило, важности признаков, вычисленные случайным лесом, являются более надежным показателем, чем важности, вычисленные одним деревом. Посмотрите на рис. 9.2

```
def plot_feature_importances_cancer(model):
n_features = cancer.data.shape[1]
plt.barh(range(n_features), model.feature_importances_, align='center')
plt.yticks(np.arange(n_features), cancer.feature_names)
plt.xlabel("Важность признака")
plt.ylabel("Признак")
plot_feature_importances_cancer(forest)
```

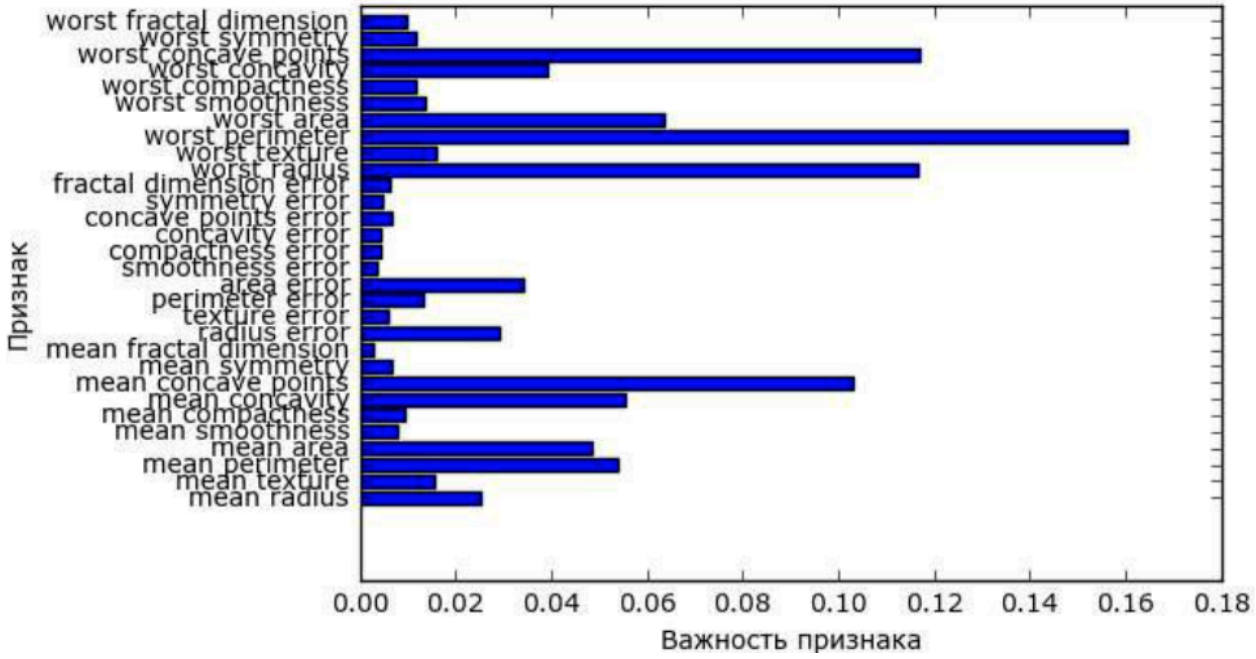

Рис. 9.2 Важности признаков, вычисленные случайным лесом для набора данных Breast Cancer

На рисунке видно, что в отличие от одиночного дерева решения случайный лес вычисляет ненулевые значения важностей для гораздо большего числа признаков. Как и дерево решений, случайный лес также присваивает высокое значение важности признаку «worst radius», однако качестве наиболее информативного признака выбирает «worst perimeter». Случайность, лежащая в основе случайного леса, заставляет алгоритм рассматривать множество возможных интерпретаций. Это приводит к тому, что случайный лес дает гораздо более широкую картину данных, чем одиночное дерево.

Преимущества, недостатки и параметры

Настоящее время случайные леса регрессии и классификации являются одним из наиболее широко используемых методов машинного обучения. Они обладают высокой прогнозной силой, часто дают хорошее качество модели без утомительной настройки параметров и не требуют масштабирования данных.

По сути случайные леса обладают всеми преимуществами деревьев решений, хотя и не лишены некоторых их недостатков. Одна из причин, в силу которой деревья решений еще используются до сих пор, – это компактное представление процесса принятия решений. Детальная интерпретация десятков или сотен деревьев нвозможна в принципе, и, как правило, деревья в случайном лесе получаются более глубокими по сравнению с одиночными деревьями решений (из-за использования подмножеств признаков). Поэтому, если вам нужно в сжатом виде визуализировать процесс принятия решений для неспециалистов, одиночное дерево решений может быть оптимальным выбором. Несмотря на то, что построение случайных лесов на больших наборах данных может занимать определенное время, его можно легко распараллелить между несколькими ядрами процессора в компьютере. Если ваш компьютер оснащен многоядерным

процессором (как почти все современные компьютеры), вы можете использовать параметр **n_jobs** для настройки количества используемых ядер. Использование большего количества процессорных ядер приведет к линейному росту скорости (при использовании двух ядер обучение случайного леса будет осуществляться в два раза быстрее), однако установка значения **n_jobs**, превышающего количество ядер, не поможет. Вы можете установить **n_jobs=-1**, чтобы использовать все ядра вашего процессора.

Вы должны помнить, что случайный лес по своей природе является рандомизированным алгоритмом и установка различных стартовых значений генератора псевдослучайных чисел (или вообще отказ от использования **random_state**) может кардинально изменить построение модели. Чем больше деревьев в лесу, тем более устойчивым он будет к изменению стартового значения. Если вы хотите получить результаты, которые потом нужно будет воспроизвести, важно зафиксировать **random_state**.

Случайный лес плохо работает на данных очень высокой размерности, разреженных данных, например, на текстовых данных. Для подобного рода данных линейные модели подходят больше. Случайный лес, как правило, хорошо работает даже на очень больших наборах данных, и обучение могут легко распараллелить между многочисленными процессорными ядрами в рамках мощного компьютера. Однако случайный лес требует больше памяти и медленнее обучается и прогнозирует, чем линейные модели. Если время и память имеют важное значение, имеет смысл вместо случайного леса использовать линейную модель.

Важными параметрами настройки являются **n_estimators**, **max_features** и опции предварительной обрезки деревьев, например, **max_depth**. Что касается **n_estimators**, большее значение всегда дает лучший результат. Усреднение результатов по большему количеству деревьев позволит получить более устойчивый ансамбль за счет снижения переобучения. Однако обратная сторона увеличения числа деревьев заключается в том, что с ростом количества деревьев требуется больше памяти и больше времени для обучения. Общее правило заключается в том, чтобы построить «столько, сколько позволяет ваше время/память».

Как было описано ранее, **max_features** случайным образом определяет признаки, использующиеся при разбиении в каждом дереве, а меньшее значение **max_features** уменьшает переобучение. В общем, лучше взять за правило использовать значения, выставленные по умолчанию: **max_features=sqrt(n_features)** для классификации и **max_features=n_features** для регрессии. Увеличение значений **max_features** или **max_leaf_nodes** иногда может повысить качество модели. Кроме того, оно может резко снизить требования к пространству на диске и времени вычислений в ходе обучения и прогнозирования.

Градиентный бустинг деревьев регресии (машины градиентного бустинга)

Градиентный бустинг деревьев регрессии – еще один ансамблевый метод, который объединяет в себе множество деревьев для создания более мощной модели. Несмотря на слово «регрессия» в названии, эти модели можно использовать для регрессии и классификации. В отличие от случайного леса, градиентный бустинг строит последовательность деревьев, в которой каждое дерево пытается исправить ошибки предыдущего. По умолчанию в градиентном бустинге деревьев регрессии отсутствует случайность, вместо этого используется строгая предварительная обрезка. В градиентном бустинге деревьев часто используются деревья небольшой глубины, от одного до пяти уровней, что делает модель меньше с точки зрения памяти и ускоряет вычисление прогнозов.

Основная идея градиентного бустинга заключается в объединении множества простых моделей (в данном контексте известных под названием слабые ученики или **weak learners**),

деревьев небольшой глубины. Каждое дерево может дать хорошие прогнозы только для части данных и таким образом для итеративного улучшения качества добавляется все большее количество деревьев.

Градиентный бустинг деревьев часто занимает первые строчки в соревнованиях по машинному обучению, а также широко используется в коммерческих сферах. В отличие от случайного леса он, как правило, немного более чувствителен к настройке параметров, однако при правильно заданных параметрах может дать более высокое значение правильности.

Помимо предварительной обрезки и числа деревьев в ансамбле, еще один важный параметр градиентного бустинга – это **learning_rate**, который контролирует, насколько сильно каждое дерево будет пытаться исправить ошибки предыдущих деревьев. Более высокая скорость обучения означает, что каждое дерево может внести более сильные корректировки и это позволяет получить более сложную модель. Добавление большего количества деревьев в ансамбль, осуществляемое за счет увеличения значения **n_estimators**, также увеличивает сложность модели, поскольку модель имеет больше шансов исправить ошибки на обучающем наборе.

Ниже приведен пример использования **GradientBoostingClassifier** на наборе данных Breast Cancer. По умолчанию используются 100 деревьев с максимальной глубиной 3 и скорости обучения 0.1:

```
from sklearn.ensemble import GradientBoostingClassifier

X_train, X_test, y_train, y_test = train_test_split( cancer.data, cancer.target,
random_state=0)

gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(gbrt.score(X_train,
y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(gbrt.score(X_test, y_test)))
```

```
Правильность на обучающем наборе: 1.000
Правильность на тестовом наборе: 0.958
```

Поскольку правильность на обучающем наборе составляет 100%, мы, вероятно, столкнулись с переобучением. Для уменьшения переобучения мы можем либо применить более сильную предварительную обрезку, ограничив максимальную глубину, либо снизить скорость обучения:

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(gbrt.score(X_train,
y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(gbrt.score(X_test, y_test)))

gbrt = GradientBoostingClassifier(random_state=0, learning_rate=0.01)
gbrt.fit(X_train, y_train)
```

```
print("Правильность на обучающем наборе: {:.3f}".format(gbrt.score(X_train,
y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(gbrt.score(X_test, y_test)))
```

```
Правильность на обучающем наборе: 0.991
Правильность на тестовом наборе: 0.972

Правильность на обучающем наборе: 0.988
Правильность на тестовом наборе: 0.965
```

Как и ожидалось, эти методы, направленные на уменьшение сложности модели, снижают правильность на обучающем наборе. В данном случае снижение максимальной глубины деревьев значительно улучшило модель, тогда как скорость обучения лишь незначительно повысило обобщающую способность.

Вновь, как и в случае с остальными моделями на основе деревьев, мы можем визуализировать важности признаков, чтобы получить более глубокое представление о нашей модели (рис. 9.3). Поскольку мы использовали 100 деревьев, вряд ли целесообразно проверять все деревья, даже если все они имеют глубину 1:

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names)
    plt.xlabel("Важность признака")
    plt.ylabel("Признак")
plot_feature_importances_cancer(gbrt)
```

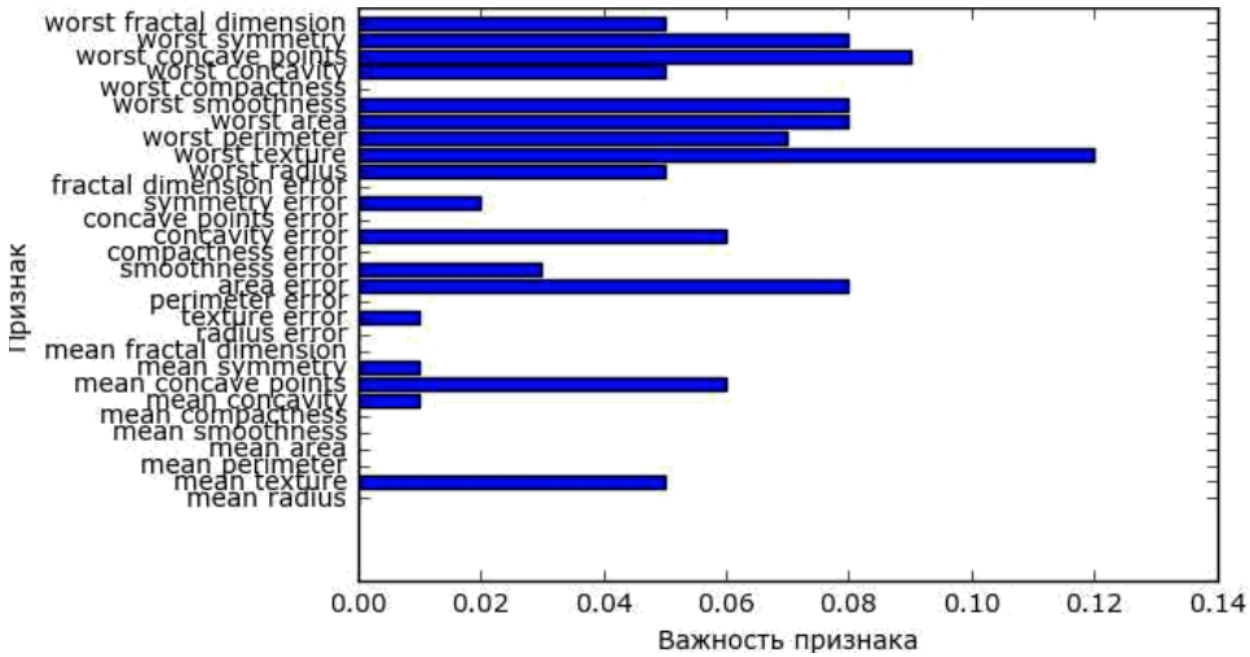

Рис. 9.3 Важности признаков, вычисленные случайным лесом для набора данных Breast Cancer

На рисунке видно, что важности признаков, вычисленные градиентным бустингом деревьев, в какой-то степени схожи с важностями признаков, полученными с помощью случайного леса, хотя градиентный бустинг полностью проигнорировал некоторые признаки.

Поскольку и градиентный бустинг и случайный лес хорошо работают на одних и тех же данных, общераспространенный подход заключается в том, чтобы сначала попытаться построить случайный лес, который дает вполне устойчивые результаты. Если случайный лес дает хорошее качество модели, однако время, отводимое на прогнозирование, на вес золота или важно выжать из модели максимальное значение правильности, выбор в пользу градиентного бустинга часто помогает решить эти задачи.

Если вы хотите применить градиентный бустинг для решения крупномасштабной задачи, возможно стоит обратиться к пакету xgboost его Python-интерфейсу, который на многих наборах данных работает быстрее (а иногда и проще настраивается), чем реализация градиентного бустинга в scikit-learn.

Преимущества, недостатки и параметры

Градиентный бустинг деревьев решений – одна из самых мощных и широко используемых моделей обучения с учителем. Его основной недостаток заключается в том, что он требуют тщательной настройки параметров и для обучения может потребоваться много времени. Как и другие модели на основе дерева, алгоритм хорошо работает на данных, представляющих смесь бинарных и непрерывных признаков, не требуя масштабирования. Как и остальные модели на основе дерева, он также плохо работает на высокоразмерных разреженных данных.

Основные параметры градиентного бустинга деревьев – это количество деревьев (n_estimators) и скорость обучения (learning_rate), контролирующая степень вклада каждого дерева в устранение ошибок предыдущих деревьев. Эти два параметра тесно взаимосвязаны

между собой, поскольку более низкое значение **learning_rate** означает, что для построения модели аналогичной сложности необходимо большее количество деревьев. В отличие от случайного леса, в котором более высокое значение **n_estimators** всегда дает лучшее качество, увеличение значения **n_estimators** в градиентном бустинге дает более сложную модель, что может привести к переобучению. Общепринятая практика – подгонять **n_estimators** в зависимости от бюджета времени и памяти, а затем подбирать различные значения **learning_rate**.

Другим важным параметром является параметр **max_depth** (или, как альтернатива, **max_leaf_nodes**), направленный на уменьшение сложности каждого дерева. Обычно для моделей градиентного бустинга значение **max_depth** устанавливается очень низким, как правило. не больше пяти уровней.

Код к лабораторной работе:

```
import sklearn
import mglearn
import matplotlib.pyplot as plt
import numpy as np



from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.25, random_state=3)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=42)
forest = RandomForestClassifier(n_estimators=5, random_state=2)
forest.fit(X_train, y_train)


fig, axes = plt.subplots(2, 3, figsize=(20, 10))
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
    ax.set_title("Дерево {}".format(i))
    mglearn.plots.plot_tree_partition(X_train, y_train, tree, ax=ax)
    mglearn.plots.plot_2d_separator(forest, X_train, fill=True, ax=axes[-1, -1], alpha=.4)
axes[-1, -1].set_title("Случайный лес")
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.show()


from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_state=0)
forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.3f}".format(forest.score(X_train, y_train)))
```

```
print("Правильность на тестовом наборе: {:.3f}".format(forest.score(X_test, y_test)))


def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names)
    plt.xlabel("Важность признака")
    plt.ylabel("Признак")
    plt.show()

plot_feature_importances_cancer(forest)


from sklearn.ensemble import GradientBoostingClassifier
X_train, X_test, y_train, y_test = train_test_split( cancer.data,
    cancer.target, random_state=0)

gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(gbrt.score(X_test, y_test)))


gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(gbrt.score(X_test, y_test)))


gbrt = GradientBoostingClassifier(random_state=0, learning_rate=0.01)
gbrt.fit(X_train, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(gbrt.score(X_test, y_test)))


gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names)
    plt.xlabel("Важность признака")
    plt.ylabel("Признак")
    plt.show()

plot_feature_importances_cancer(gbrt)
```

```
print("---------- Part 2 ----------")


from sklearn.datasets import make_blobs
X, y = make_blobs(centers=4, random_state=8)
y = y % 2

mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
plt.show()


from sklearn.svm import LinearSVC
linear_svm = LinearSVC().fit(X, y)

mglearn.plots.plot_2d_separator(linear_svm, X)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
plt.show()


X_new = np.hstack([X, X[:, 1:] ** 2])
from mpl_toolkits.mplot3d import Axes3D, axes3d
figure = plt.figure()
ax = Axes3D(figure, elev=-152, azim=-26)
mask = y == 0

ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b', cmap=mglearn.cm2, s=60)
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r', marker='^', cmap=mglearn.cm2,
s=60)
ax.set_xlabel("признак0")
ax.set_ylabel("признак1")
ax.set_zlabel("признак1 ** 2")
plt.show()


linear_svm_3d = LinearSVC().fit(X_new, y)
coef, intercept = linear_svm_3d.coef_.ravel(), linear_svm_3d.intercept_
figure = plt.figure()

ax = Axes3D(figure, elev=-152, azim=-26)

xx = np.linspace(X_new[:, 0].min() - 2, X_new[:, 0].max() + 2, 50)
yy = np.linspace(X_new[:, 1].min() - 2, X_new[:, 1].max() + 2, 50)

XX, YY = np.meshgrid(xx, yy)
ZZ = (coef[0] * XX + coef[1] * YY + intercept) / -coef[2]
```

```
ax.plot_surface(XX, YY, ZZ, rstride=8, cstride=8, alpha=0.3)
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b', cmap=mglearn.cm2, s=60)
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r', marker='^', cmap=mglearn.cm2,
s=60)

ax.set_xlabel("признак0")
ax.set_ylabel("признак1")
ax.set_zlabel("признак1 ** 2")
plt.show()


ZZ = YY**2
dec = linear_svm_3d.decision_function(np.c_[XX.ravel(), YY.ravel(), ZZ.ravel()])
plt.contourf(XX, YY, dec.reshape(XX.shape), levels=[dec.min(), 0, dec.max()],
    cmap=mglearn.cm2, alpha=0.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
plt.show()


from sklearn.svm import SVC
X, y = mglearn.tools.make_handcrafted_dataset()
svm = SVC(kernel='rbf', C=10, gamma=0.1).fit(X, y)
mglearn.plots.plot_2d_separator(svm, X, eps=.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
sv = svm.support_vectors_
sv_labels = svm.dual_coef_.ravel() > 0

mglearn.discrete_scatter(sv[:, 0], sv[:, 1], sv_labels, s=15, markeredgewidth=3)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
plt.show()



fig, axes = plt.subplots(3, 3, figsize=(15, 10))

for ax, C in zip(axes, [-1, 0, 3]):
    for a, gamma in zip(ax, range(-1, 2)):
        mglearn.plots.plot_svm(log_C=C, log_gamma=gamma, ax=a)

axes[0, 0].legend(["class 0", "class 1", "sv class 0", "sv class 1"], ncol=4, loc=(.9, 1.2))
plt.show()


X_train, X_test, y_train, y_test = train_test_split( cancer.data, cancer.target, random_state=0)
svc = SVC()
svc.fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.2f}".format(svc.score(X_train, y_train)))
```

```
print("Правильность на тестовом наборе: {:.2f}".format(svc.score(X_test, y_test)))
plt.show()


plt.plot(X_train.min(axis=0), 'o', label="min")
plt.plot(X_train.max(axis=0), '^', label="max")
plt.legend(loc=4)
plt.xlabel("Индекс признака")
plt.ylabel("Величина признака")
plt.yscale("log")
plt.show()


min_on_training = X_train.min(axis=0)
range_on_training = (X_train - min_on_training).max(axis=0)
X_train_scaled = (X_train - min_on_training) / range_on_training
print("Минимальное значение для каждого признака\n{}".format(X_train_scaled.min(axis=0)))
print("Максимальное значение для каждого признака\n {}".format(X_train_scaled.max(axis=0)))


X_test_scaled = (X_test - min_on_training) / range_on_training
svc = SVC()
svc.fit(X_train_scaled, y_train)
print("Правильность на обучающем наборе: {:.3f}".format( svc.score(X_train_scaled, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(svc.score(X_test_scaled, y_test)))


svc = SVC(C=1000)
svc.fit(X_train_scaled, y_train)
print("Правильность на обучающем наборе: {:.3f}".format( svc.score(X_train_scaled, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(svc.score(X_test_scaled, y_test)))
```